Transient-Fault Recovery for Chip Multiprocessors

Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz
School of Electrical and Computer Engineering, Purdue University
{gomaa, cscarbro, vijay, pomeranz}@ecn.purdue.edu

Abstract

To address the increasing susceptibility of commodity chip multiprocessors (CMPs) to transient faults, we propose Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR). CRTR extends the previously-proposed CRT for transient-fault detection in CMPs, and the previously-proposed SRTR for transient-fault recovery in SMT. All these schemes achieve fault tolerance by executing and comparing two copies, called leading and trailing threads, of a given application. Previous recovery schemes for SMT do not perform well on CMPs. In a CMP, the leading and trailing threads execute on different processors to achieve load balancing and reduce the probability of a fault corrupting both threads; whereas in an SMT, both threads execute on the same *processor. The inter-processor communication required to compare the threads introduces latency and bandwidth problems not present in an SMT.*

To hide inter-processor latency, CRTR executes the leading thread ahead of the trailing thread by maintaining a long slack, enabled by asymmetric commit. CRTR commits the leading thread before checking and the trailing thread after checking, so that the trailing thread state may be used for recovery. Previous recovery schemes commit both threads after checking, making a long slack suboptimal. To tackle inter-processor bandwidth, CRTR not only increases the bandwidth supply by pipelining the communication paths, but also reduces the bandwidth demand. By reasoning that faults propagate through dependences, previously-proposed Dependence-Based Checking Elision (DBCE) exploits (true) register dependence chains so that only the value of the last instruction in a chain is checked. However, instructions that mask operand bits may mask faults and limit the use of dependence chains. We propose Death- and Dependence-Based Checking Elision (DDBCE), which chains a masking instruction only if the source operand of the instruction dies after the instruction. Register deaths ensure that masked faults do not corrupt later computation. Using SPEC2000, we show that CRTR incurs negligible performance loss compared to CRT for inter-processor (one-way) latency as high as 30 cycles, and that the bandwidth requirements of CRT and CRTR with DDBCE are 5.2 and 7.1 bytes/cycle, respectively.

1 Introduction

Technology scaling trends that lead to smaller and faster transistors and lower supply voltages result in increased susceptibility to transient faults and degraded reliability even in commodity microprocessors. To utilize the high transistor counts afforded by technology scaling, the microprocessor industry is adopting chip multiprocessors (CMPs) (e.g., the IBM Power 4 is a four-processor CMP). CMPs are building blocks for server-class machines for which reliability is a key concern. To address reliability issues in CMPs, [6] briefly describes the Chip-level Redundantly Threaded multiprocessor (CRT) for transient-fault *detection*. In this paper, we propose hardware-assisted transient-fault *recovery* for CMPs.

Simultaneously and Redundantly Threaded (SRT) processors [9] and Simultaneously and Redundantly Threaded processors with Recovery (SRTR) [17] are proposals for transient-fault detection and recovery, respectively, based on Simultaneous Multithreaded (SMT) processors [16]. SRT and SRTR, and other proposals [11,14], provide fault tolerance by replicating an application into two communicating threads, one (called the leading thread) executing ahead of the other (called the trailing thread), and by comparing their values. SRT maintains a long *slack* (e.g., 256 instructions) between the threads so that the trailing thread can use memory load values and branch outcomes of the leading thread to avoid memory latencies and mispredicted computations. SRT commits register values *before* checking for faults but guarantees fault detection and avoids memory corruption by checking stores before commit. To achieve recovery, SRTR commits register values (in either thread) only *after* the values are checked. Consequently, a long slack causes leading thread stalls. At the same time, a short slack causes trailing thread stalls. SRTR solves this dilemma by using a moderate slack (e.g., 32 instructions) and reducing trailing thread stalls by exploiting leading instructions' complete-to-commit times.

CRT applies SRT's detection to CMPs. However, extending the CMP-based CRT to provide recovery by naively repeating the SMT-based SRTR extension of SRT does not achieve high performance. There is a key difference between CMP- and SMT-based schemes: In a CMP, the leading and trailing threads execute on *different* processors to achieve load balancing and reduce the probability of a fault corrupting both threads [6]; whereas in an SMT, both threads execute on the *same* processor. Because of layout constraints, the processors in a CMP cannot be physically close. The inter-processor communication required to compare the values from the threads makes the latency and bandwidth of the communication paths critically important. These issues are not addressed by SRTR.

Compared to a CMP, CRT and CRTR need extra hardware queues (like SRT's and SRTR's queues) to hold the communicated values, and extra inter-processor wires for the communication. The global wires are bound to impose both a substantial latency (e.g., 20 cycles) and a limited bandwidth (e.g., 20 bytes per cycle) on the communication. Typical complete-to-commit times are insufficient to hide the resultant delay for *every* register and memory value, rendering SRTR's slack inadequate for CMPs. Although [6] does not discuss this issue, this delay is not a problem for CRT: Because CRT is a detection scheme and commits register values before checking, CRT can employ a long slack without stalling the leading thread.

We propose Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR). Because the inter-processor delay fundamentally requires a slack longer than that used for SRTR, we use *asymmetric commit*, a departure from SRTR's strategy of not committing before checking. CRTR enables long slack by allowing the leading thread to commit register updates *before* checking, so that long slacks do not hold up leading thread commits. However, CRTR allows the trailing thread to commit register updates only *after* checking, so that the register state of the trailing thread can be used for recovery. In contrast, CRT allows *both* threads to commit register updates *before* checking, eliminating the possibility of recovery using the trailing thread. While AR-SMT [11] used asymmetric commit as a recovery strategy in SMT without explicit latency considerations, CRTR uses it to hide inter-processor latency by enabling a long slack. As in CRT, CRTR commits memory updates (i.e., stores) only after checking, so that memory is guaranteed to be correct. Because stores are less frequent than register updates, CRTR can increase the slack without stalling leading thread commits. Upon detecting a fault, CRTR raises an exception and copies the trailing register state to the leading thread.

The asymmetric commit hides inter-processor latency. To tackle inter-processor bandwidth requirements, we pipeline the inter-processor paths and hide the latency of the pipelining using the asymmetric commit. While this pipelining boosts the bandwidth supply, we reduce the bandwidth demand by employing two techniques. First, while SRTR checks speculative values, CRTR, like CRT, communicates and checks only committed values. Second, we extend the SRTR scheme of Dependence-Based Checking Elision (DBCE). By reasoning that faults propagate through dependences, DBCE exploits (true) register dependence chains so that *only* the last instruction in a chain is checked. Earlier instructions in the chains in *both* threads completely elide communication and checking, reducing bandwidth pressure. DBCE redundantly builds chains in both threads and checks its own functionality.

DBCE encounters problems with masking instructions, which may mask a fault in its inputs by producing the correct output even if an input is faulty (e.g., *r2 := r1 & 0xff00*, *r1 := (r2 < r3)*). Such masking violates the key assumption of DBCE that faults are propagated by dependences. A later instruction in the chain of a masking instruction cannot detect the masked fault, and an irrecoverable error ensues if the faulty value is committed and consumed by some later computation. Consequently, SRTR suggests disallowing masking instructions from joining DBCE chains. Because many integer and almost all floating-point instructions (due to their finite precision) are masking, this restriction on masking instructions limits the DBCE chain lengths and reduces the effectiveness of DBCE.

We extend DBCE to exploit the death of register values, and propose Death- and Dependence-Based Checking Elision (DDBCE). The problem with a masking instruction occurs if a source operand is faulty, and some later instruction, *other than* the masking instruction, also consumes the faulty value. By tracking register death, we identify those masking instructions that are the *last* consumers of their source operands—i.e., the source operands die after the masking instruction. The operand death ensures that any masked fault does not corrupt later computation, allowing masking instructions to join chains without loss of recovery. Because many register values are consumed by only one or two instructions, DDBCE boosts the bandwidth reduction of DBCE.

The main contributions and results of this paper are:

- To tackle inter-processor latency, we use asymmetric commits.
- To tackle inter-processor bandwidth, we not only increase the bandwidth supply by pipelining the communication paths, but also reduce the bandwidth demand (1) by extending the previously-proposed DBCE to DDBCE, and (2) by checking only committed values and not speculative values.
- Using SPEC2000, we find that CRTR incurs negligible performance loss compared to CRT for inter-processor (one-way) latency as high as 30 cycles.
- Our results show that the bandwidth requirements for CRT, CRTR without DBCE, CRTR with conservative DBCE, and CRTR with DDBCE are 5.2, 9.8, 7.8, and 7.1 bytes/cycle, respectively.

CRTR is guaranteed to provide recovery from single transient faults except those that affect the (non-ECC-protected) register file, in which case CRTR guarantees detection.

In Section 2, we discuss related work. We review CRT in Section 3. We describe CRTR in Section 4 and DDBCE in Section 5. In Section 6, we present experimental results, and conclude in Section 7.

2 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 [11]. AR-SMT and its follow-up for CMPs, called Slipstream[14], also propose using speculation techniques to allow communication of data values and branch outcomes between the leading and trailing threads to accelerate execution. SRT improves on AR-SMT via the two optimizations of slack fetch and checking only stores [9]. SRTR extends SRT to provide recovery for SMT [17]. The CRT paper explores design options for fault detection via multithreading, and briefly discusses detection on CMPs [6].

AR-SMT and Slipstream briefly mention recovery using their equivalent of the trailing thread in SMT and CMP, respectively. However, even though both CRTR and Slipstream target CMPs, they are fundamentally different in the following three ways: (1) Slipstream may not recover from some faults. (2) Slipstream does not address a central correctness issue for CMPs: memory locations may be modified by another processor (e.g., during multiprocessor synchronization) between the time the leading thread loads a value and the time the trailing thread tries to load the same value. CRTR uses the previously-proposed Load Value Queue [9] to communicate the leading load values to the trailing thread. (3) Slipstream allows the leading thread to commit to memory before checking, requiring two copies of memory. Doubling the memory size may stress the memory hierarchy and degrade performance. In contrast, CRTR needs only one copy of memory because it checks stores and commits only one store to memory. CRTR tackles the latency

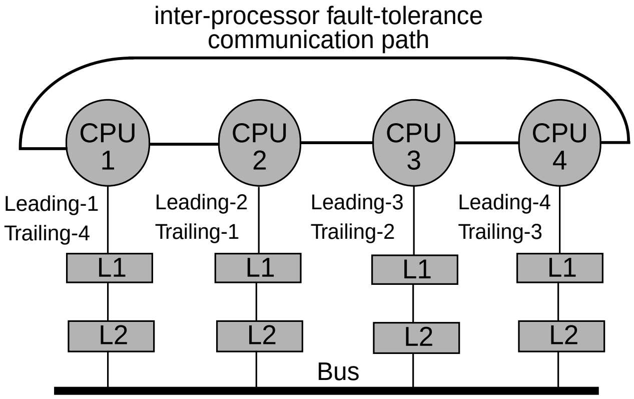


FIGURE 1: CRT and CRTR.

and bandwidth problems that are ignored by Slipstream: the communication of load values and store check confirmations.

Software recovery schemes such as [3,12], which use hardware detection, need (hardware or software) checkpointing of program state (memory and registers), incurring considerable performance cost even when there are no faults. In contrast, CRTR does not require any checkpointing, and it involves raising an exception and copying the register state (but not memory state) from the trailing thread only when faults are detected.

Another paper proposes hardware recovery using superscalar hardware without any SMT support [8]. DIVA is a 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 [10, 7, 4, 13].

The Compaq NonStop Himalaya [3] and IBM z900 (formerly S/390) [12] provide fault 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 components disagree, execution is 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.

3 Transient-fault detection in CMPs

The Chip-level Redundantly Threaded multiprocessor (CRT) provides transient-fault detection using a chip multiprocessor (CMP). CRT borrows the detection scheme from the SMT-based Simultaneously and Redundantly Threaded (SRT) processors and applies the scheme to CMPs. CRT replicates 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 CRT. As mentioned in Section 1, CRT executes the leading and trailing threads on different processors to achieve load balancing and to reduce the probability of a fault corrupting both threads. In Figure 1, we show a 4-CPU CRT running leading thread i on CPU i, and trailing thread i on CPU i+1 (modulo 4).

Because detection is based on replication, the extent to which the application is replicated is important. CRT replicates register values (in the register file of each processor) but not memory values. CRT's leading thread commits stores only after checking, so that memory is guaranteed to be correct. CRT 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 for detection; other instructions commit *without* checking. CRT uses a store buffer (StB) in which the leading thread places its committed store values and addresses. The store values and addresses of the trailing thread are compared against the StB entries to determine whether a fault has occurred. Only one copy of the checked store reaches the cache hierarchy. Because data in the cache hierarchy is not replicated, other forms of protection such as ECC are needed for the cache hierarchy.

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. CRT allows only the leading thread to access the cache and uses the Load Value Queue (LVQ) to hold the leading load values and addresses. The trailing thread loads from the LVQ instead of repeating the load from the cache, after comparing load addresses to ensure that no fault has occurred. The Active Load Address Buffer proposed in [9] is an alternative for the LVQ that also addresses this problem. CRT uses the simpler LVQ.

A key optimization in the SMT-based 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. In [6], the authors use a line predictor queue, instead of the BOQ, to allow the leading thread to control the trailing thread's fetch in the case of SMT. The slack and the communication of branch outcomes hide the memory latencies of the leading thread and avoid 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. However, this scheme works only in an SMT where both threads fetch from the same i-cache, but it is not applicable to a CMP. [6] does not explain how to extend the scheme to CMPs.

CRT assumes that uncached accesses are performed non-speculatively. CRT synchronizes uncached accesses from the leading and trailing threads, compares the addresses, and replicates the load data. CRT assumes that code does not modify itself, although self-modifying code in regular CMPs 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, CRT suggests forcing the threads to the same execution point and then delivering the interrupt synchronously to both threads.

CRT communicates values from the processor running the leading thread to the one running the trailing thread. Compared to a CMP, CRT needs extra hardware queues (LVQ and BOQ) to hold the communicated values, and it needs extra inter-processor wires for the communication. To minimize the number of wires, we place the trailing threads on the processor adjacent to that running the leading thread, as shown in Figure 1.

Although [6] does not mention this issue, the inter-processor communication in CRT does not pose a problem, as mentioned in Section 1. The inter-processor latency is mostly absorbed by the long slack, and the bandwidth pressure is tolerable because CRT communicates only branch outcomes, load values and store values, but not register values.

4 Transient-fault recovery for CMPs

We propose *Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR)* which enhances CRT to include transient-fault recovery. Like CRT, CRTR assumes SMT processors in the CMP and uses the configuration illustrated in Figure 1. Unlike CRT, CRTR must not allow *any* trailing instruction to commit before it is checked for faults, so that the register state of the trailing thread may be used for recovery. However, the leading thread in CRTR may commit register state before checking, as in CRT. This asymmetric commit strategy allows CRTR to employ a long slack to absorb inter-processor latencies. As in CRT, CRTR commits stores only after checking. Because stores are relatively infrequent, the slack can be increased without stalling leading thread commits.

CRTR uses the long slack to hide the inter-processor communication latency between the leading and trailing threads. In addition to communicating branch outcomes, load addresses, load values, store addresses, and store values like CRT, CRTR also communicates register values. CRTR employs sender-initiated (i.e., leading-thread initiated) communication and queues up the values at the processor running the trailing thread. Thus, if the slack is appropriately long, the values of the leading thread reach the trailing thread before it needs the values, despite incurring the communication delay.

4.1 Slack fetch

We modify the instruction fetch in CMP to check whether the leading and trailing threads are separated by at least the amount of a pre-specified threshold; if they are, the trailing thread is allowed to fetch, otherwise the leading thread is allowed to fetch. SRT and SRTR implement this scheme by modifying the SMT ICOUNT [15]. ICOUNT maximizes the number of independent instructions in the pipeline by fetching from the thread that has fewer instructions waiting for their source operands, implying more independence. SRT and SRTR skew the count of waiting instructions by the difference between the current slack and the threshold, so that the leading (trailing) thread is allowed to fetch if the current slack is less (more) than the threshold.

Unlike SRT and SRTR where the leading and trailing threads of the *same* program execute on a processor, in CRTR two unrelated threads—the leading thread of *one* program and the trailing thread of *another* program—execute on one processor, as shown in Figure 1. Consequently, maintaining slack between the threads on a processor is meaningless. Recall from Section 3 that using a line predictor queue to control fetching, as proposed in [6], is not applicable to CMPs.

Instead, CRTR employs the following policy: of the two threads on the processor, if the slack of one is such that it should fetch (i.e., the slack of the leading thread with respect to *its* trailing thread is below the threshold, and the reverse condition for the trailing thread) and the slack of the other is such that it should not fetch, then there is no conflict and the first thread fetches. If neither thread should fetch (i.e., the slack of the leading thread is above the threshold with respect to its trailing thread, and the reverse for the trailing thread), or both threads should fetch, then CRTR breaks the tie by using ICOUNT.

The implementation of the CRTR policy needs to address the following point. Because the leading and trailing threads execute on different processors, accurately estimating the current slack is difficult. The main issue is that every cycle CRTR needs to know the separation between the leading and trailing threads, which are on different processors. To address this issue, CRTR counts the leading thread slack in terms of the number of waiting stores that have not been confirmed to commit by the checker and the number of values (branch outcomes, load values and register values) queued due to limited inter-processor bandwidth. For the trailing thread slack, CRTR counts the number of values (branch outcomes, load values, store values, and register values) waiting to be consumed. Thus, CRTR approximates the separation in committed instructions (which cannot be determined locally in one processor) by the number of waiting instructions and values (which can be determined within each processor). Interestingly, our scheme performs better than counting slack in terms of number of instructions separating the leading and trailing threads, as we show in Section 6.1.

4.2 Sending leading values: Need for SRTR’s RVQ

Like CRT, the leading thread in CRTR communicates committed values to the trailing thread for branch outcomes, load addresses, load values, store addresses, and store values. CRTR additionally communicates committed register values. Consequently, the values are not affected by mispredictions. The leading thread sends its values in commit (i.e., program) order, and the trailing thread consumes the values in commit order.

Sending values at commit eliminates the need for the values to be cleaned up on mispredictions, which is needed in SRTR (SRTR stores values and checks at completion to exploit complete-to-commit times [17]). However, there is an implementation difficulty raised by sending values at commit: register values are written back to the register file at instruction completion, and the instruction does not have the value at commit. Therefore, a leading instruction has to retrieve the register value from the register file to send the value to the trailing thread. Similarly, the trailing thread has to retrieve the value from the register file to perform the check at instruction commit. Such retrievals would add significantly to the bandwidth pressure on the already-belabored register file [17].

SRTR avoids extra bandwidth pressure on the register file by using the register value queue (RVQ) to hold register values for checking. CRTR borrows the idea to place register values in the RVQ at writeback. Both leading and trailing threads deposit their values in their respective RVQs at writeback, and the leading thread RVQ entries are communicated to the trailing thread. SRTR then retrieves leading thread register values from the RVQ when the trailing instruction completes, whereas CRTR retrieves values for checking at trailing thread commit. See the

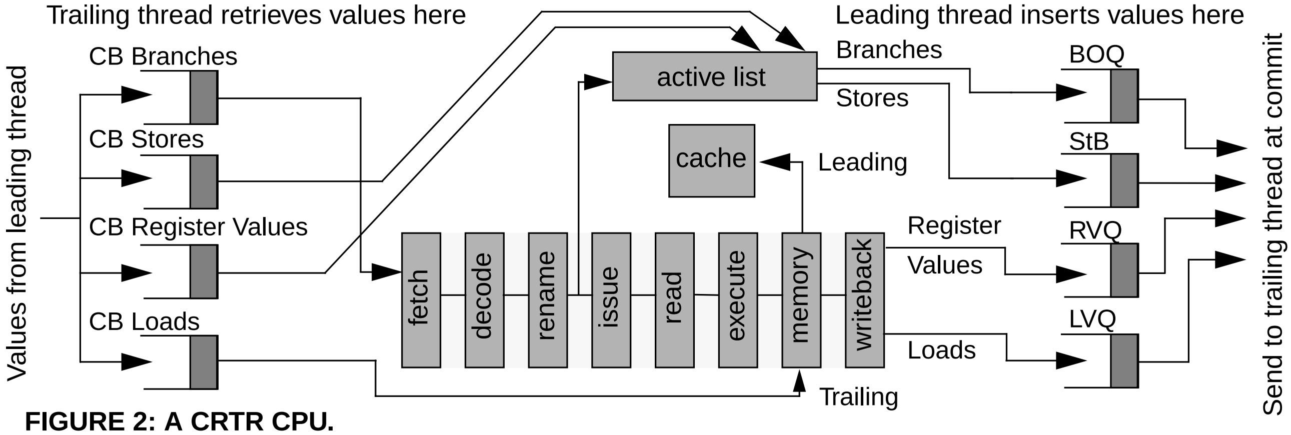


FIGURE 2: A CRTR CPU.

RVQ and LVQ shown in Figure 2.

SRTR proposes Dependence-Based Checking Elision (DBCE) to reduce the bandwidth pressure on the RVQ itself. By reasoning that faults propagate through dependences, DBCE exploits (true) register dependence chains so that *only* the last instruction in a chain is checked. Earlier instructions in the chains in *both* threads completely elide checking, reducing RVQ bandwidth pressure. While CRTR can also use DBCE, we do not pursue the topic of RVQ bandwidth reduction further because SRTR covers the topic in detail [17]. In Section 5, we address the communication bandwidth pressure between leading and trailing threads by extending the DBCE.

Sending or checking at commit is not a problem for branch outcomes and store values because these are available at commit. Branches hold their outcomes to update the branch predictor counters at commit (even if the history register is updated speculatively, the counters are updated at commit); and stores are sent from the store buffer to the cache at commit. See the StB and BOQ shown in Figure 2.

4.3 Matching leading value to trailing instruction

The committed leading values, load addresses, load values, store addresses, store values, and branch outcomes need to be queued at the trailing thread because the trailing thread may not be ready to consume the values as soon as they arrive. They are queued in separate queues of the *check buffer (CB)* (i.e., register, load, store, and branch values are held separately). See the CB shown in Figure 2. Upon reaching commit, the trailing instructions check their values against the head of the appropriate CB, in a strict queue order.

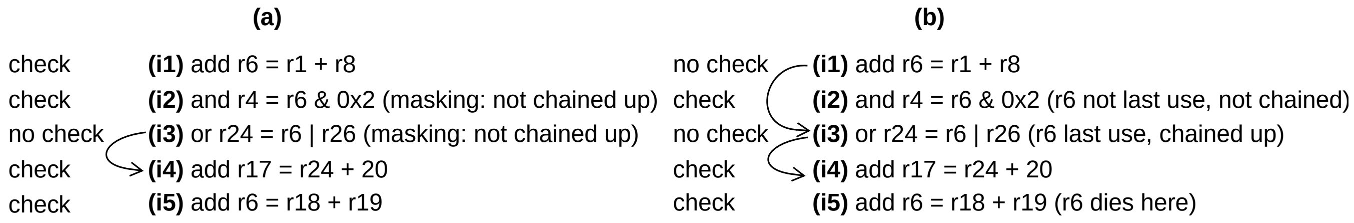
As in CRT, the branch outcomes of the leading thread are used by the trailing thread as predictions. If there are no faults, the trailing thread would never encounter a branch misprediction. If the outcome of the leading thread is incorrect due to a fault, the resolution of the trailing branch flags a misprediction which triggers a transient-fault exception.

Checking in commit order raises the issue of matching leading thread values to the correct trailing instruction. Branch outcomes, store values and register values are straightforward. Trailing thread fetch is sequential and empties the branch CB in queue order, and therefore, the correct trailing branch matches the branch outcome. Similarly, stores and register values are checked at trailing commit, and they can be emptied in queue order and matched to the correct trailing instruction. If there is a mismatch due to a fault, the mismatch causes the affected instructions to fail their check, and a transient-fault exception is raised.

Unfortunately, load values are more difficult to match with the correct trailing loads. Unlike branch, store, and register values, which are needed at commit, load values are needed at execution. Because execution is out of order, trailing loads may issue out-of-order with respect to each other. This problem also exists for CRT and SRT. SRT advocates restricting issue of trailing loads to be in program order, which requires special cases in the instruction scheduler. In order to tackle this issue, we propose a different approach which allows out-of-order issue of trailing loads. This approach relies on the fact that the trailing thread—in the absence of faults—does not mispredict and commits all of the dispatched instructions. Therefore, by using a counter, each load can know its corresponding load CB entry number *at dispatch*. The counter matches the ith trailing load with the ith load CB entry (modulo load CB size).

If a fault occurs, a trailing load could possibly access an incorrect entry in the load CB, yet the load addresses could match and retrieve an incorrect value. This situation could occur in two ways: (1) If a branch outcome from the leading thread is corrupted, then a subsequent trailing load could match incorrectly. However, the fault that corrupted the outcome will be detected, causing the incorrectly-matched load to squash. (2) A fault could corrupt the load-matching counter, resulting in an incorrect load match. Because loads are not checked, such an incorrectly-matched load would commit, leading to an irrecoverable fault. To avoid this problem, we use a self-correcting counter (which uses ECC-like encoding).

Upon reaching the load CB, each trailing thread load checks its address against that of its corresponding load CB entry. If this entry is valid and the addresses match, the load retrieves the value and completes. If the load CB entry is invalid, the load CB signals a “miss” to the pipeline and the load stays pending in the issue queue to be reissued later, similar to an L1 miss. The only difference is that a load missing in L1 completes whenever the miss is satisfied. Here, hitting an invalid entry implies that the CB entry has not yet arrived from the leading thread, as long as there are no faults. Accordingly, the pending load may be reissued if any new load entries arrive from the leading thread. If a fault has corrupted the address of the pending load in either thread, the load will not match its corresponding valid load CB entry. In that case, such a load would



```
(a)
check      (i1) add r6 = r1 + r8
check      (i2) and r4 = r6 & 0x2 (masking: not chained up)
no check   (i3) or r24 = r6 | r26 (masking: not chained up)
check      (i4) add r17 = r24 + 20
check      (i5) add r6 = r18 + r19

(b)
no check   (i1) add r6 = r1 + r8
check      (i2) and r4 = r6 & 0x2 (r6 not last use, not chained)
no check   (i3) or r24 = r6 | r26 (r6 last use, chained up)
check      (i4) add r17 = r24 + 20
check      (i5) add r6 = r18 + r19 (r6 dies here)
```

FIGURE 3: An example of (a) conservative DBCE, and (b) DDBCE

eventually reach the commit point in the active list, and a transient-fault exception would be raised. The load CB is ECC-protected, so its contents are not vulnerable to faults.

4.4 Committing leading stores

In CRTR, just as in CRT, the only communication from the trailing thread back to the leading thread is the result of checking stores, so that leading stores may commit. The trailing thread commits its store as soon as the check is performed, avoiding the overhead of any acknowledgment of receipt from the leading thread. Therefore, it is important to ensure that the check result is not corrupted on its way to the leading thread. Consequently, CRTR sends the check results to the leading thread under ECC protection.

Because of the inter-processor communication delay and the slack, leading stores wait for trailing stores to be completed and the checking to be performed. In modern processors, because the StB is searched by loads to honor memory dependences, the StB cannot be made large. Consequently, there is some pressure on the StB of the leading thread, as pointed out in [6]. On one hand, a long slack helps hide inter-processor delays and branch and memory delays; on the other hand, the pressure on the StB increases with a long slack. We set the slack to balance hiding of latencies and pressure on the StB.

4.5 Recovery using the trailing thread state

The trailing processor preserves the faulting instruction PC so that execution can restart from that PC value. The exception handler saves the trailing register state and PC to the CMP shared memory and launches a “restoring thread” on the leading processor to load the saved register state and PC value from memory. To ensure that faults do not corrupt the saving or restoring processes themselves, the restoring thread redundantly saves the register and PC state loaded in the leading processor to a different set of memory locations. The handler then compares those locations with the trailing processor state. If the comparison fails, the saving and restoring are redone.

The cost of the exception and register copying is low enough to allow acceptable recovery times (e.g., less than 10-40ms of network round-trip delays so that recovery time is imperceptible for networked clients).

There are faults from which CRTR 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, CRTR cannot recover from this fault. To avoid this loss of recovery, one solution is to provide ECC on the register file, as suggested for SRTR [17].

5 Tackling inter-processor bandwidth

The asymmetric commit hides inter-processor latency. To tackle inter-processor bandwidth requirements, we pipeline the inter-processor paths and hide the latency of the pipelining using the asymmetric commit. The inter-processor wires shown in Figure 1 are split into several segments with latches between the segments. While this pipelining boosts the bandwidth supply, we reduce the bandwidth demand by employing two techniques. First, unlike SRTR which checks speculative values, CRTR, like CRT, communicates and checks only committed values. Second, we extend SRTR’s Dependence-Based Checking Elision (DBCE). While SRTR uses DBCE to reduce the RVQ bandwidth, we extend DBCE to reduce the inter-processor register communication bandwidth of CRTR. It is important to note that DBCE reduces only register bandwidth and does *not* impact communication due to loads, stores, and branches.

5.1 Applying SRTR’s DBCE to CRTR

By reasoning that faults propagate through dependences, DBCE exploits (true) register dependence chains so that *only* the last instruction in a chain is checked. For example, in Figure 3(a), *i3* and *i4* form a chain, and *i4* is checked. To keep the implementation simple, we follow SRTR and 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). Earlier instructions in the chains in *both* threads completely elide communication and checking, reducing bandwidth pressure. 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. Then, a transient-fault exception is raised. 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.

The SRTR paper explains the implementation of DBCE in detail. Therefore, we give an abstracted description of the implementation and point out the key differences. DBCE consists of (1) forming dependence chains in the leading thread and the corresponding chains in the trailing thread, (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.

DBCE uses a hardware queue called the dependence chain queue (DCQ) to hold instructions and determine dependences by matching appropriate register operands. DCQ forms the chains (identical chains in both threads), and records the chain information in the Check Table for later use. The chain formation occurs at rename, and upon completion, instructions look up the Check Table to determine whether they need to check. A non-last instruction waits at commit for the last instruction of its chain to check. When the checking occurs, the last instruction uses the Check Table to signal the other instructions in the chain. At commit, each instruction ensures that the last instruction in its chain is identical to the last instruction in the chain of its counterpart instruction; otherwise a fault is signaled.

There are many differences between SRTR's DBCE implementation and CRTR's. First, because SRTR's leading instruction can commit only after checking, SRTR's chains need to be short (e.g., 3-4 instructions) so that the first instruction in a leading chain does not unduly wait for the last instruction in the trailing chain. Because CRTR's leading thread commits before checking and because CRTR's slack is long, CRTR's chains can be longer. If a DBCE chain is *m* instructions long, DBCE checks only one of the *m* instructions, reducing register value bandwidth. Therefore, longer chains are better.

Second, because SRTR checks speculative instructions, DBCE forms chains using speculative instructions. If chains are allowed to cross branches, branch mispredictions will require clean-up. To avoid such clean-ups, SRTR disallows chains from crossing branches, resulting in short chains. In contrast, CRTR communicates committed values and applies DBCE to non-speculative instructions which do not include mispredictions. Hence, CRTR allows chains to cross branches, encouraging longer chains. That is, an instruction producing a value before a branch and an instruction consuming the value after the branch may participate in a chain.

Third, because SRTR's DBCE reduces RVQ bandwidth demand, and loads and stores do not use the RVQ in SRTR, these instructions do not participate in SRTR chains. In CRTR, however, we want to reduce the bandwidth demand, which is partly due to loads and stores. Therefore, CRTR's chain may end in a load or a store. Because load values are *always* needed by the trailing thread, and stores do not produce any register values, a load or store can only be the last instruction of a chain. Any such chain ending in a load or a store does not require any register value bandwidth (as opposed to a chain that ends in an ALU instruction and needs to communicate the last instruction's register value) because load and store values are communicated *anyway.*

Fourth, while SRTR forms both leading and trailing chains at rename, CRTR forms leading chains at commit and trailing chains at rename. Because the trailing thread does not mispredict in the absence of faults, the leading and trailing chains are guaranteed to be identical, even though the chains are formed at different points in the pipeline. Any faulty branch outcome will cause the leading and trailing chains to mismatch and trigger a fault exception. While the sending of leading values occurs at commit after chains are formed, the delay due to chain formation is absorbed by CRTR's long slack. The checking of trailing values occurs at commit after trailing chains are completely formed. Because trailing chains are formed at rename, any delay in trailing chain formation is absorbed by the time gap between rename and commit.

5.2 DDBCE

DBCE must consider masking instructions, which produce the same output value for different input values (e.g., *r4 := r6 & 0x2* in Figure 3(a)). Some instructions produce the same output value for different input values only if one of their inputs assumes a specific value (e.g., *r24 := r6 | r26* when *r26* is all 1's). We conservatively consider all such instructions as masking. A masking instruction may mask a fault on its inputs by producing the correct output even if an input is faulty. Such masking violates the key assumption of DBCE that faults are propagated by dependences. The last instruction in a chain that includes a masking instruction cannot detect the masked fault, and an irrecoverable error ensues if the faulty value is committed and consumed later. For instance, if *i1* and *i2* of Figure 3(a) are chained, the check done on the value of *r4* does not cover all the bits of *r6*. If *i1* produces a faulty value of *r6* so that only the bits masked by *i2* are faulty, *r4*'s check will not detect the fault and will cause the chain to commit. If *r6* is used by instructions later than those shown, recovery will not be possible. Moreover, prior to *i1* in Figure 3(a), if there is an instruction *r1 := r16 + 17* and this instruction produces a faulty value for *r1*, the fault could propagate through *r6* and be masked in *r4*. If a chain contains this instruction, and *i1* and *i2* are committed, a later instruction consuming *r1* may detect an irrecoverable error.

DBCE disallows masking instructions to form chains, with the exception that a masking instruction may start a chain. A masking instruction is allowed at the beginning of a chain because the source operands of the instruction will be checked in previous chains, without allowing any faults to be masked. In Figure 3(a), *i2* and *i3* are masking; and neither is chained up to *i1*, but *i3* starts a chain with *i4*.

Many integer and almost all floating-point instructions (due to their finite precision) are masking. For instance, in *f1 := f2 + f3*, if *f3* has a small value and *f2* has a large value, then the addition may mask faults in *f3*. We consider all logical and shift operations except xor, xnor, and not; all floating-point instructions; all direct and indirect jumps (calls and returns); and all branches as masking instructions. We consider loads and stores as non-masking (but they need to be communicated). Integer multiply and divide are masking.

Integer add is non-masking because of the following: Assume that two integers, *a* and *b*, are added, and assume there is a fault in one of them (single fault assumption). Without loss of generality assume that *b* has a fault and we denote the faulty *b* as *b'*. From the properties of addition, *a+b* and *a+b'* cannot be equal. The only way in which a fault can produce a fault-free sum is if *a* and *b* are stored in the *same* register (and *a* = *b*). In this case, one of the (faulty or fault-free) additions may result in an overflow while the other does not. Because ALUs detect overflow, we compare the sums and the overflow conditions of the leading and trailing adds to avoid the masking of any fault in an integer addition.

Conservatively restricting masking instructions to the start of chains exposes the bandwidth pressure. To address this problem, we extend DBCE to exploit the death of register values, and propose Death- and Dependence-Based Checking Elision (DDBCE). The problem with a masking instruction occurs if one of its source operands is faulty, and some later instruction, *other than* the masking instruction, also consumes the faulty value. By tracking register death, we identify those masking instructions that are the *last* (in program order) consumers of their source operands—i.e., the source operands die after consumption by the masking instruction. The operand death ensures that any masked fault does not corrupt later computation, allowing masking instructions to join chains without loss of recovery. In Figure 3(b), masking instruction *i3* is the last use of *r6* (before *r6* dies in *i5*). *i3* can be chained up to *i1* because any fault in *r6* is not visible beyond *i3*. The resulting chain includes *i1*, *i3*, and *i4*. Masking instruction *i2* is not the last use of *r6* and cannot guarantee that faulty *r6* will not be used later, and therefore it is not chained. Because many register values are consumed by only one or two instructions, DDBCE boosts DBCE's bandwidth reduction.

The extent of the reduction in bandwidth by using DDBCE is affected by the last instruction of each chain. For instance, a DDBCE chain can end in a branch. Such a chain does not require any register value bandwidth, because the branch outcome is sent anyway. This point is similar to that of DBCE chains ending in loads or stores, as described in Section 5.1.

5.2.1 Implementation simplification

Chaining of masking instructions introduces a subtle implementation difficulty. Consider an instruction sequence *a1*, *a2*, and *a3* in program order (with *a1* being the earliest). If *a3* is a masking instruction that writes into a register *r3* and *a3* is chained with *a2* through *r2* (i.e., *a2* writes to *r2* and *a3* reads *r2*), then *r2* must die at *a3* (i.e., *a3* is the last use of *r2*). *r2*'s death guarantees that any fault in *r2* masked by *a3* is not visible to later computation. In addition, if *a2* is chained to *a1* through *r1*, then *r1* must die after *a3*. *r1*'s death is needed because faults in *r1* propagating to *r2* may be masked by *a3*, resulting in a fault-free *r3* value. However, later uses of *r1* will result in detectable but irrecoverable errors once *a1* commits the faulty *r1*. Therefore, in order for a masking instruction to join a chain, all destination registers in the chain must die after the masking instruction. This observation is true whether or not *a1* and *a2* are masking instructions.

The above observation implies the need to check every instruction in a chain for register death in order to add a masking instruction. For implementation simplicity, we prefer to avoid the need to check for deaths of multiple registers when a single instruction is added to a chain. Therefore, we allow a masking instruction to chain only to an unchained, non-masking preceding instruction or to any (chained or unchained) masking preceding instruction. A preceding masking instruction need not be unchained because all of the destinations of the preceding chain must be dead for the preceding chain to include a masking instruction. Therefore, lengthening the chain by an additional masking instruction requires checking the death of only one register. Following a masking instruction, we allow any number of non-masking instructions to be chained because non-masking instructions propagate faults and do not require register deaths. Using regular expressions and denoting a masking instruction by *m* and a non-masking instruction by *n*, we allow chains of the form nm^*n^* or m^*n^*.

DDBCE extends the DCQ to identify register death. In DBCE, a non-masking instruction entering the DCQ searches the DCQ to see if the instruction is dependent on some previous instruction. DDBCE uses the same scheme as DBCE for non-masking instructions. For masking instructions, death must be established. An incoming instruction writing to an architectural register kills the previous value bound to the register. In conventional renaming, the previous physical register mapped to an instruction's architectural destination register is known to the instruction, so that the previous register may be freed upon the instruction's commit. Accordingly, the incoming instruction searches the DCQ and marks the last instruction which uses the previous physical register as a source.

When a register *r* which is a source of a masking instruction *m* dies, DDBCE checks whether it is valid to link a chain C1, that ends with an instruction *i* producing *r*, with a chain C2 that starts with *m*. For the linking to be valid, the resulting chain must be legal, and the chain is checked as follows: every non-masking instruction that attaches itself to a chain accepts a flag indicating the number of instructions already in the chain. The flag indicates one instruction, or more than one instruction (as per DBCE rules, only the first instruction may be masking). If C1 includes two or more instructions, then C1 is not linked with C2; otherwise, every linking is allowed.

Implementing linking of two chains has two additional components. The first component is the chaining of *i* and *m*, and marking *i* as a non-last instruction. The second component involves the chain starting with *m*. While in formation in the DCQ, chains are tagged by the first instruction in the chain. In the Check Table, however, after the chains are completely formed, chains are tagged by the last instruction so that the non-last instructions can know to commit when the last instruction is checked. The linking of the two previously-formed chains needs retagging of the entire second chain by the first instruction of the first chain. This retagging amounts to parallel writing of the tags of the second chain.

DDBCE relies on searches through the DCQ to form legal chains and establish death. To keep DDBCE implementable, DCQ's size needs to be restricted. In Section 6.4, we show that a DCQ with 20 or 30 entries suffices.

6 Experimental results

We modify the Simplescalar out-of-order simulator [2] to model a CMP environment. Each core in the CMP is an SMT processor. Each core has its own private L1 and L2 caches. Table 1 shows the configuration parameters used for each core. Because the StB in real systems is typically 20-30 entries, we assume an StB of 20 entries in order to model pressure on the StB as discussed in [6] and in Section 4.4. We model the system bus which is a split-transaction, pipelined bus connecting the private L2 caches to memory. We model the details of request and response phases of the transactions.

In most cases, we show results for a 2-CPU CMP. However, we show 4-CPU runs for our most important result: comparison of CRTR to CRT. Because neither CRT nor CRTR uses any shared resource for fault-tolerance support, 2-CPU and 4-CPU behavior should be similar. The results for CRT and CRTR comparison show that 2-CPU and 4-CPU behavior are nearly identical. We do not show 4-CPU results anywhere else in the interest of simulation time and because of this similarity. We do not show runs with one thread per CPU because such a run is

Table 1: Hardware parameters for base system.

Component	Description
Issue	SMT, 4-way out-of-order issue, 128-entry RUU
Branch prediction	8k hybrid of bimodal and gshare, 16-entry RAS, 4-way 1K BTB (10-cycle misprediction penalty)
L1 I- and D-cache (private per CPU)	64KB, 32-byte blocks, 4-way, 2-cycle hit, lock-up free
L2 unified cache (private per CPU)	1 Mbyte,64-byte blocks, 4-way, 12-cycle hit
Main memory	Infinite capacity, 100 cycle latency
System bus	Split transaction, bus clock speed to CPU clock speed ratio = 1:2
BOQ/LVQ/StB	96/128/20 entries

not representative of a CMP workload and does not stress per-CPU resources.

Based on Section 4.1, CRTR's slack is defined by two thresholds—the leading threshold (x) and the trailing threshold (y) —and denoted as "x/y". On the leading thread side, when the total number of unconfirmed stores in the StB and values (branch outcomes, load values, and register values) queued due to limited inter-processor bandwidth exceeds the leading threshold, the leading thread *stops* fetching because the slack has grown to its limit. On the trailing thread side, when the total number of values (branch outcomes, load values, store values, and register values) waiting to be consumed in the CB queues exceeds the trailing threshold, it *starts* fetching because the slack is too low.

For our 2-CPU runs, we pair up two SPEC2000 benchmarks to generate the CMP workload. We choose pairs from combinations of low-IPC and high-IPC programs. Table 2 presents the pairings used in our simulation. For each experiment, we fast-forward 2 billion instructions per benchmark. We then run until all the programs complete at least 400 million instructions. All benchmarks use ref inputs. We show results for 20 out of the 26 SPEC2000 applications. We do not show the other 6 benchmarks because they take a long time to simulate. Our metric for performance is instruction throughput measured in instructions committed per cycle by all the CPUs in the CMP.

We present results in the absence of faults in order to study the performance cost of CRTR versus CRT. Faults are expected to be rare enough that the overall performance will be determined by fault-free behavior. In addition, our simulator does not support exception handling required for CRTR.

We begin by comparing CRT with SRTR to show that SRTR performs poorly on CMPs because the inter-processor communication latency is too high to be hidden by complete-to-commit times. We then compare CRTR to CRT under varying slack thresholds. This comparison shows that CRTR incurs negligible performance loss compared to CRT. Next, we vary the communication latency to show that CRTR's performance does not degrade for a wide range of expected latencies. Finally, we show the communication bandwidth reduction achieved by DBCE with conservative chaining and by DDBCE. We also show the impact of bandwidth on CRTR performance using conservative DBCE and using DDBCE.

Table 2: Benchmarks.

Benchmark set	set #	IPC pairing	2-CPU IPC
ammp+galgel	1	FP/FP low/high	2.20
fma3d+equake	2	FP/FP low/low	1.62
gap+sixtrack	3	Int/FP high/high	2.27
gcc+vortex	4	Int/Int low/low	1.46
gzip+parser	5	Int/Int high/low	1.91
perlbmk+swim	6	Int/FP low/low	1.89
twolf+mgrid	7	Int/FP high.high	2.45
vpr+crafty	8	Int/Int high/high	2.26
wupwise+mesa	9	FP/FP high/high	2.34
mcf+eon	10	Int/Int low/high	2.51

6.1 CRT versus SRTR

In Figure 4, we compare CRT with SRTR, which is a recovery scheme for SMT. Because the line-predictor queue for controlling fetch in [6] does not apply to CMPs (as mentioned in Section 4.1), we assume a fixed slack for CRT. In addition, we implement CRT using our fetch policy of slack thresholds. We refer to these two systems as "CRT-fixed" and "CRT." We show performance normalized to the base CMP without any fault tolerance. The benchmark set numbers used by the X-axis labels are from Table 2. In this experiment, we assume a 10-cycle latency and infinite bandwidth for the inter-processor communication. CRT-fixed uses a slack of 256 instructions as per [6], and CRT uses a slack threshold of 16/32. CRT-fixed and CRT use the same sizes for BOQ/LVQ/StB shown in Table 1. SRTR uses a slack of 40 instructions, and a predQ/LVQ/StB of 128/128/20 entries, comparable to or larger than CRT-fixed and CRT. The slack for CRT-fixed, CRT, and SRTR are the best-performing values.

CRT-fixed and CRT are on average 19% and 15% worse than the base CMP, respectively. We see that CRT, which uses only local information to implement slack, performs a little better than CRT-fixed. Compared to the base CMP, CRT's degradation comes from two factors: StB fill-ups and the instruction overhead of the trailing thread (the base CMP executes only one copy of a program) competing for the CPU resources. Some programs incur large slowdowns (e.g., *set 7 "twolf+mgrid"* and *set 8 "vpr+crafty"*) due to frequent filling of the StB. The rest of the degradation is due to the instruction overhead.

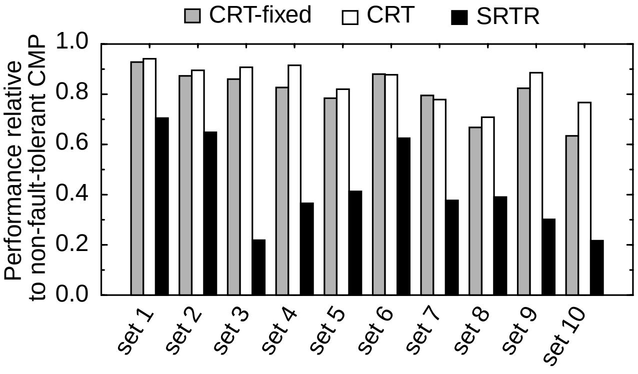


FIGURE 4: CRT vs. SRTR.

SRTR's performance on a CMP is on average 58% worse than the base CMP. SRTR's leading thread does not commit any instruction until it is checked by the trailing thread. SRTR exploits complete-to-commit times of leading instructions such that by the time the leading instruction commits, the trailing instruction has completed and the check has been performed. The complete-to-commit times already hide SRTR's slack. Because typical complete-to-commit times are 20-30 cycles [17], the l0-cycle latency on *every* load, store, branch and register value, and another 10 cycles for check confirmation, cannot be hidden by the remainder of the complete-to-commit times. Like CRT, extra instructions due to running two copies of a program also contribute to SRTR's degradation.

From these results, we can see that SRTR is not suitable for a CMP even at inter-processor latencies of 10 cycles and infinite bandwidth. With wire delays getting relatively worse with every generation, SRTR's performance is bound to worsen.

6.2 CRT versus CRTR

In Figure 5, we compare CRTR with CRT. As before, we show performance normalized to the base CMP without any fault tolerance. CRT's slack threshold is 16/32, which is its best-performing threshold. We vary CRTR's slack threshold as 16/8, 32/16, and 64/32. In this experiment, we assume a 20-cycle latency (one-way) and infinite bandwidth for the inter-processor communication. In the previous section, we used a latency of 10 cycles to show that even at 10 cycles SRTR does not perform well, but we believe that a 20-cycle latency for global inter-processor wires is more appropriate.

From Figure 5, we see that CRTR using a threshold of 32/16 performs close to CRT. The reasons for this similarity are: (1) In terms of the overhead of StB fill-ups and extra instructions, CRT and CRTR are similar. (2) Both CRT and CRTR communicate branch and memory values. The inter-processor latency of 20 cycles is seen by this communication in both CRT and CRTR. CRTR's long slack, enabled by asymmetric commits, absorbs this latency to a similar extent as CRT's long slack. (3) Although CRTR additionally communicates register values, CRTR performance is not degraded because we assume infinite inter-processor bandwidth in this experiment (we show finite bandwidth in Section 6.4). Because loads and stores together are frequent (e.g., loads and stores are 30-50% of all instructions), the communication latency of register values is hidden under those of load values and store check confirmations. Therefore, CRTR incurs negligible performance degradation compared to CRT.

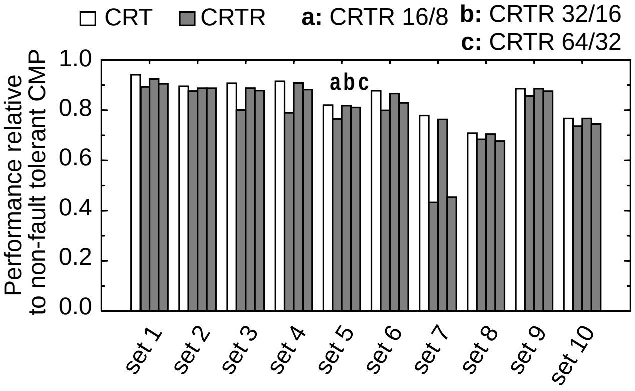


FIGURE 5: CRT vs. CRTR.

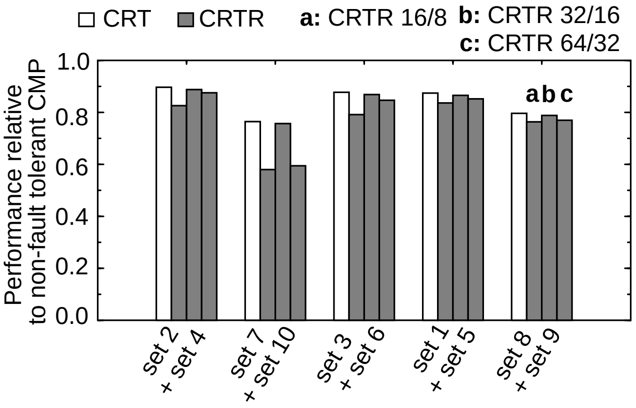


FIGURE 6: CRT vs. CRTR: 4 CPUs

We also see that the shorter slack threshold of 16/8 slows down CRTR in many cases. With the shorter slack threshold, the leading thread prematurely stops fetching because the thread frequently hits the threshold. This stopping causes the communication latency to be exposed. When the slack threshold is 64/32, *set 7* (*twolf+mgrid)* degrades in performance because the trailing thread does not fetch until it accumulates the large threshold, slowing down the rate of checking. This slowdown causes the leading thread's StB to fill up, stopping commits.

In Figure 6, we compare CRTR with CRT using a 4-CPU CMP with 20-cycle latency and infinite bandwidth for communication. As in the 2-CPU case, CRTR using a threshold of 32/16 performs close to CRT. The performance of the programs is similar to that in the 2-CPU case, confirming our claim in the beginning of Section 6 that two and four CPUs behave similarly.

6.3 Inter-processor latency

In this section, we study the effect of the communication latency on CRTR and CRT performance. The CRT paper uses 4-cycle inter-processor latency in its study, and our evaluation extends that study. As before, we show performance normalized to the base CMP without any fault tolerance. We vary the latency as 20, 30, and 60 cycles (one-way) for both CRT and CRTR. We still assume infinite inter-processor bandwidth.

Figure 7 shows that increasing the inter-processor latency from 20 cycles ("a" bars) to 30 cycles ("b" bars) has little impact on CRT and CRTR, with the average degradation with respect to the base CMP increasing from 15% to 16% for both. CRTR continues to perform as well as CRT even at 30-cycle latency. As explained in Section 6.2, load and store communication present in both CRT and CRTR impacts their performances more than register value communication impacts CRTR. CRTR's register communication is hidden under load value and store check confirmations.

At 60-cycle latency ("c" bars), both CRT and CRTR incur an average performance degradation of 22% compared to the base CMP. In this case, the StB fills up frequently due to the 60-cycle delay for the store values to be sent and another 60-cycle delay for the confirmation to return to the leading thread. Because

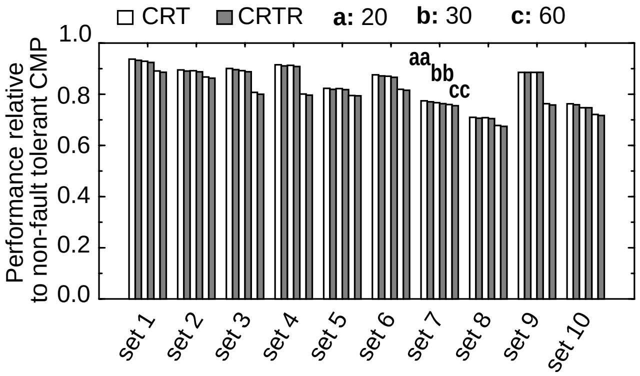


FIGURE 7: Effect of inter-processor latency

such fill-ups occur for both CRT and CRTR, CRTR falls only slightly (< 1%) behind CRT, even at 60-cycle latency.

The long slack enabled by asymmetric commit makes CRTR tolerant to inter-processor communication latency of even 30 cycles. This tolerance is important for CRTR to be effective in future technologies in which global wire delays pose a serious problem for microprocessor and CMP performance.

6.4 Inter-processor bandwidth

Finally, we study the effect of inter-processor bandwidth on CRTR. First, we present the bandwidth required by CRT and by CRTR using DBCE and using DDBCE. Then, we present the impact of finite bandwidth on CRTR performance. The SRTR paper [17] allows optimistic chaining of masking instructions in its study, and our evaluation with conservative DBCE (i.e., without chaining masking instructions) extends that study.

In Table 3, we compare CRT, CRTR with no DBCE, CRTR with conservative DBCE, and CRTR with DDBCE. We compute the bandwidth requirements for each technique by averaging across the bandwidth requirements of the individual programs. We also show the minimum and maximum values across the programs. The table also shows the impact of varying the DCQ size as 20, 30, and 60 entries for DBCE and DDBCE.

From the table, we see that while CRT communicates about 5.2 bytes/cycle, CRTR with no DBCE almost doubles the bandwidth at 9.8 bytes/cycle. DBCE using a 30-entry DCQ cuts down CRTR's bandwidth requirement to 7.8 bytes/cycle, a reduction of about 20%. DDBCE, also using a 30-entry DCQ, further reduces the requirement to 7.1 bytes/cycle, which is a reduction of 9% over DBCE. Note that although we consider all floating-point instructions as masking, floating-point programs have non-masking integer instructions (e.g., address calculators, loop indices) which chain and reduce bandwidth even under DBCE.

Looking at the figures for DCQ size variation, we see that a 20-entry DCQ is within 95% of a 30-entry DCQ for both DBCE and DDBCE. A 60-entry DCQ provides only an additional 3% reduction. These numbers show that a DCQ size as small as 20 entries captures most of the benefit of DBCE and DDBCE. This small size makes the issues discussed in Section 5.2 tractable in a real DCQ implementation.

Table 3: Bandwidth requirements in bytes/cycle

	CRT	CRTR	DBCE			DDBCE		
			20	30	60	20	30	60
Max	6.1	12.2	9.6	8.1	8.0	8.6	7.4	7.4
Min	4.2	7.4	7.1	6.8	6.8	6.7	6.6	6.5
Avg	5.2	9.8	8.2	7.8	7.6	7.4	7.1	6.9

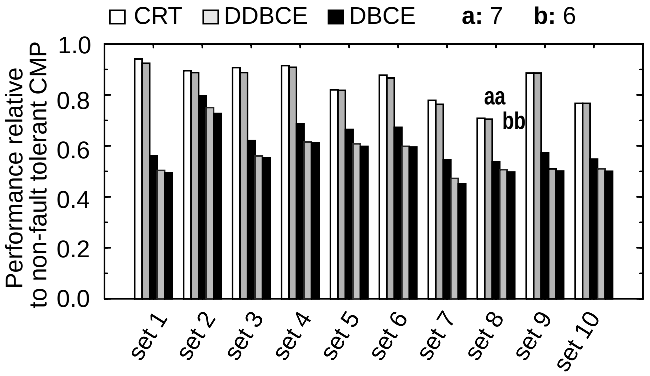


FIGURE 8: Effect of inter-processor bandwidth

In Table 4, we show average chain length—using a DCQ of size 30 instructions—for DBCE ignoring masking, conservative DBCE which does not chain masking instructions, and DDBCE. The numbers in Table 3 and Table 4 do not correlate linearly because of the following observations: (1) The total bandwidth is due to load/store/branch values and register values. DBCE and DDBCE target only the register value bandwidth component. Therefore, a chain length of m does not imply a reduction in total bandwidth by a factor of m. Moreover, an increase in average chain length by a factor of x% (going from DBCE to DDBCE) does not imply a reduction in total bandwidth of x%. (2) The last instruction of each chain further affects the extent to which the register value bandwidth is reduced. For instance, a DBCE chain may end in a load (Section 5.1). Such a chain does not contribute anything to the register value bandwidth because the load value is sent anyway, whereas a chain ending in a register ALU operation contributes one register value to the register value bandwidth. Similarly, a DDBCE chain may end in a branch (Section 5.2). Such a chain does not require any register value bandwidth, because the branch outcome is sent anyway.

In Figure 8, we plot CRT, CRTR with DDBCE and conservative DBCE using a restricted bandwidth of 7 bytes/cycle ("a" bars), and 6 bytes/cycle ("b" bars). We pick these bandwidth points based on Table 3. We want to show the performance penalty of using DBCE at DDBCE's required bandwidth of 7 bytes/cycle; and the penalties of using DBCE and DDBCE at a bandwidth of 6 bytes/cycle, which is lower than DDBCE's required bandwidth. We do not show CRTR with no DBCE since its bandwidth requirements are even higher. We use a latency of 20 cycles, and as before, we show performance normalized to the base CMP without any fault tolerance.

CRTR with DDBCE using 7 bytes/cycle performs as well as CRT, which needs only 5 bytes/cycle. However, CRTR with

Table 4: Average chain length

Benchmarks	Ignore Masking	DBCE	DDBCE
INT	2.31	1.55	1.62
FP	2.17	1.35	1.43

DBCE using 7 bytes/cycle incurs a degradation of 25% compared to CRTR with DDBCE. This substantial degradation justifies extending DBCE with DDBCE for improved filtering of fault-tolerance traffic. This degradation occurs because DBCE is unable to filter traffic to this restricted bandwidth, causing the queues (LVQ, BOQ, and StB) to fill up. At a bandwidth of 6 bytes/cycle, which is below the level to which DDBCE can filter traffic, both DDBCE and DBCE incur large performance degradations. This graph shows that for CRTR to perform as well as CRT, the net cost is the higher inter-processor bandwidth requirement of CRTR.

7 Conclusion

We proposed Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) for chip multiprocessors (CMPs). CRTR extends the previously-proposed CRT for transient-fault detection in CMPs, and the previously-proposed SRTR for transient-fault recovery in SMT. To hide inter-processor latency, CRTR uses a long slack enabled by asymmetric commit and uses the trailing thread state for recovery. To tackle inter-processor bandwidth, CRTR both increases the bandwidth supply by pipelining the communication paths, and reduces the bandwidth demand (1) by extending the previously-proposed Dependence-Based Checking Elision (DBCE) to Death- and Dependence-Based Checking Elision (DDBCE), and (2) by checking only committed values and not speculative values. By reasoning that faults propagate through dependences, DBCE exploits (true) register dependence chains so that only the value of the last instruction in a chain is checked. However, instructions that mask operand bits may mask faults and limit the use of dependence chains. DDBCE chains a masking instruction only if the source operand of the instruction dies after the instruction. Register deaths ensure that masked faults do not corrupt later computation.

Using SPEC2000, we found the following: (1) Because both CRT and CRTR communicate load and store values, the latency of the additional register value communication in CRTR is hidden under the load and store value communication, allowing CRTR to perform as well as CRT. However, the additional communication has a cost: while CRT needs 5.2 bytes/cycle, CRTR with DDBCE needs a higher 7.1 bytes/cycle. (2) The long slack enabled by asymmetric commit makes CRTR tolerant to inter-processor latency of even 30 cycles. This tolerance is important for CRTR to be effective in future technologies in which global wire delays will pose a serious problem for microprocessor and CMP performance. (3) The bandwidth requirements for CRT, CRTR without DBCE, CRTR with conservative DBCE, and CRTR with DDBCE are 5.2, 9.8, 7.8, and 7.1 bytes/cycle, respectively. Because inter-processor bandwidth is a key resource in present-day and future CMPs, the traffic reductions achieved by DBCE and DDBCE are important.

Acknowledgements

The work of T. N. Vijaykumar was supported in part by NSF CAREER award No. CCR-9875960 and NSF Instrumentation grant No. CCR-9986020. The work of I. Pomeranz was supported in part by NSF Grant No. CCR-0049081.

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] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002.