Computer Architecture Lab at Carnegie Mellon (CALCM) Technical Report 2008-2

REDAC: Distributed, Asynchronous Redundancy in Shared Memory Servers

Brian T. Gold[1], Babak Falsafi[1,2], James C. Hoe[1], and Ken Mai[1]

[1]Computer Architecture Lab (CALCM), Carnegie Mellon University, Pittsburgh, PA, USA
[2]Parallel Systems Architecture Lab (PARSA), Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
http://www.ece.cmu.edu/~truss

Abstract

The emergence of multi-core architectures—driven by continued technology scaling—has led to concerns about increasing soft- and hard-error rates in commodity designs. Because modern chip designs consist of multiple high-speed clock domains, conventional lockstepped redundant execution is no longer practical. Recent work suggests an asynchronous approach to redundant execution, where processor pairs independently execute an instruction stream and treat any differences like soft errors, invoking rollback recovery. Because prior designs buffer instruction results within the out-of-order instruction window, they are limited to tightly coupled redundancy within a single chip, which limits availability and serviceability in the presence of hard errors.

We propose REDAC, a set of lightweight mechanisms for distributed, asynchronous redundancy within a shared-memory multiprocessor. REDAC provides scalable buffering for unchecked state updates, permitting the distribution of redundant execution across multiple nodes of a scalable shared-memory server. The REDAC mechanisms achieve high performance by enabling speculation across common serializing instructions and mitigating the effects of input incoherence. We evaluate REDAC using cycle-accurate full-system simulation of common enterprise workloads and show that performance overheads average just 10% when compared to a non-redundant system. These results are comparable to the performance of a similarly configured lockstep design, but offer the substantial benefits of asynchronous redundancy.

1 Introduction

As data processing and storage requirements continue to grow, vendors of business-critical computing systems rely increasingly on shared-memory platforms to build high-performance, scalable servers that run existing application and system software. Unfortunately, the increasing levels of integration that drive scalability also result in rising soft- and hard-error rates [6,24]. Recent work [3,16,20,23,30,31,32] has focused on tightly coupled redundant execution, where instructions are executed redundantly within a single chip. Although tightly coupled redundancy mitigates soft error rates, availability and serviceability are limited in the event a hard error occurs.

To provide strong reliability, availability, and serviceability guarantees, systems must form redundant pairs across chips. Conventional approaches to cross-chip, distributed redundancy rely on lockstepping of the redundant pairs [4,34]. Because of the move to high-frequency CMP designs with multiple, asynchronous clock domains, maintaining lockstep is increasingly difficult and costly to engineer [5,19].

Proposed alternatives to lockstep execution rely on high-bandwidth structures such as the load-value queue (LVQ) [31], which explicitly guarantees redundant executions observe an identical view of memory. These mechanisms support *synchronous* redundancy, where the redundant instruction streams are precisely replicated; however, bandwidth limitations preclude scaling LVQ structures to cross-chip redundant execution. Prior proposals [15] for guaranteeing synchronous redundancy across chips have instead relied on lockstep guarantees to replay only off-chip load values.

In contrast, recent work [37] proposed *asynchronous* redundancy, where replicated processors are permitted to access the memory system independently. Although originally proposed for tightly coupled redundant pairs, we observe that any system supporting asynchronous redundancy must address three challenges:

- **Buffering State Updates.** Because the redundant executions are asynchronous, the processor must buffer state updates until the redundant pair has compared the results.
- **Serializing Instructions.** The system must support serializing operations such as memory ordering operations, programmed I/O accesses, special (non-renamed) register accesses, and external interrupt delivery.
- **Input Incoherence.** Because asynchronous redundant executions access the memory system independently, they may observe different values for the same dynamic load—what [37] refers to as *input incoherence*—which requires rollback recovery to correct. The effect of these rollbacks must be mitigated by limiting the frequency of input incoherence and the associated recovery cost.

The Reunion [37] design constructs asynchronous redundant pairs from the cores on a tightly coupled CMP and leverages existing speculation support within the out-of-order instruction window to buffer instructions pending comparison. These mechanisms do not scale sufficiently to

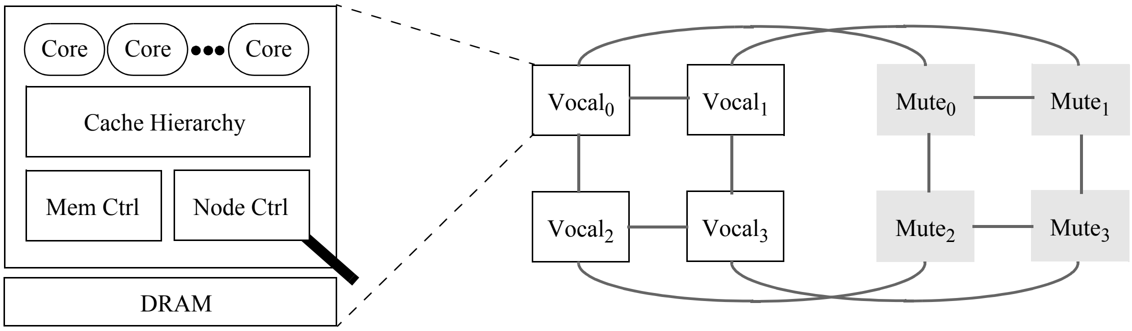


FIGURE 1. Example 8-node *REDAC* system. Each node consists of an identical CMP and a portion of the physical memory. All of the cores in Vocal$_i$ are paired with the cores in Mute$_i$.

support cross-chip, distributed redundancy. With less bandwidth and longer latencies for comparing redundant executions, the processor must effectively buffer large numbers (e.g., hundreds or thousands) of instructions, which far outstrips the available speculation resources in the out-of-order instruction window. Because of the longer checking latency, frequent serializing instructions such as memory ordering operations become a major performance bottleneck. Finally, because the redundant executions now include larger L2 or L3 caches, the rate of input incoherence increases as stale values are kept for much longer periods of time, and the cost of subsequent recovery increases as more instructions must be replayed and one or more off-chip accesses are required.

In this paper, we propose *REDAC*[1], a set of mechanisms that enables distributed, asynchronous redundancy in scalable shared-memory servers with little performance overhead. Figure 1 illustrates the basic architecture of *REDAC*. We evaluate *REDAC* using cycle-accurate full-system simulation of common enterprise workloads and show that performance overheads average just 10% when compared to a non-redundant system. These results are comparable to the performance of a similarly configured lockstep design, but offer the substantial benefits of asynchronous redundancy. We make the following specific contributions in support of *REDAC*:

- **Scalable Result Buffering.** We show that prior proposals for long-range speculation provide sufficient buffering of unchecked state updates for distributed, asynchronous redundancy. Our evaluations demonstrate that with as few as two hardware checkpoints, performance overhead in *REDAC* averages just 15% (25% in worst case), while four checkpoints reduce the average overhead to 10% (17% in worst case).
- **Speculative Serializing Operations**. We show that prior mechanisms [37] cannot efficiently support speculation across common memory ordering instructions. We describe an approach to integrate existing speculation mechanisms [9,14,43] with redundant execution and demonstrate that memory order speculation reduces stalls in *REDAC* by up to 40%.
- **Limited Input Incoherence.** We propose mechanisms to mitigate the impact of input incoherence in *REDAC*. *REDAC* limits input incoherence to data races by providing *invalidation hints* to the redundant execution, which reduces stalls by up to 70% in our study. Furthermore, we observe that because most data races occur in or around memory ordering and atomic operations, *REDAC* can limit the cost of recovery by shortening the window of speculation around these instructions. Our evaluations show a drastic reduction in stalls—over 2X in several workloads—with this technique.

Paper Outline. In Section 2, we present background on our fault model and the challenges of distributed, asynchronous redundancy. Section 3 describes the proposed *REDAC* mechanisms. We evaluate *REDAC* in Section 4, present related work in Section 5 and conclude in Section 6.

2 Background: Redundant Computation

2.1 System and Fault Models

Increasing levels of integration, diminishing node capacitance, and lower noise margins in high-performance architectures have led to growing concerns about rising error rates from a variety of transient, permanent, and intermittent faults [6,22,41]. Our fault model assumes transient faults, such as those arising from cosmic rays or alpha particles, manifest as bit flips in latches, embedded memory cells, or incorrect logic results, but leave the underlying hardware operational. Intermittent and permanent faults, whether due to failing transistors or damaged wires [41], may result in fail-silent operation (no further results following error), fail-fast operation (an error detected and reported immediately), or—in the worst case—silent data corruption that goes undetected.

[1] REdundant Distributed Asynchronous Checking

Redundant computation, as we propose in this paper, provides protection for the processor datapath without requiring modifications to the complex and timing-critical pipeline itself. Instruction results that have yet to undergo output comparison do not require additional protection, but we must protect pending state updates which are being checked or have already been checked. We assume information codes protect cache arrays and critical communication buses [34].

The mechanisms we propose in this paper reduce the vulnerability of scalable shared-memory architectures, typified by the system of Figure 1. In such designs, a portion of the physical memory space is collocated with each processor chip. We assume, as is common practice in industry today [11,17], that each chip contains the necessary interface logic to access the local memory and to handle network traffic for coherent memory access from remote processor and memory nodes. Although our evaluation in Section 4 uses directory-based coherence protocols, the *REDAC* mechanisms apply to snooping protocols as well.

The non-compute portions of the system—DRAM, I/O, and interconnect—are protected through combinations of information redundancy (e.g., memory mirroring or parity [28], RAID [27], and ECC [34]) and link- or protocol-level retry mechanisms (e.g., [39]). Control and interface logic is assumed to be protected through hardening or robust circuit design [34].

As in prior proposals [37] for tightly coupled asynchronous redundancy, our design compresses architectural state updates into a fingerprint signature [38]. Fingerprints enable low-bandwidth comparisons while capturing essentially every instruction result. Because the fingerprint uses a hash-based signature, it is subject to collisions that can leave errors undetected or uncorrected. However, designers can strengthen the hash design to meet desired error budgets while maintaining acceptable comparison bandwidth [36].

2.2 Challenges of Distributed, Asynchronous Redundancy

To enable asynchronous, software-transparent redundancy in shared-memory systems, Smolens et al. [37] proposed the Reunion execution model and presented a design that forms redundant pairs from tightly coupled cores on a CMP. The enabling observation of Reunion is that, in the absence of data races and errors, replicas will execute an identical instruction stream. The rare case where execution diverges is treated as a soft error, causing rollback and re-execution. Upon rollback, a *synchronizing request* ensures that replicas observe identical values and guarantees forward progress.

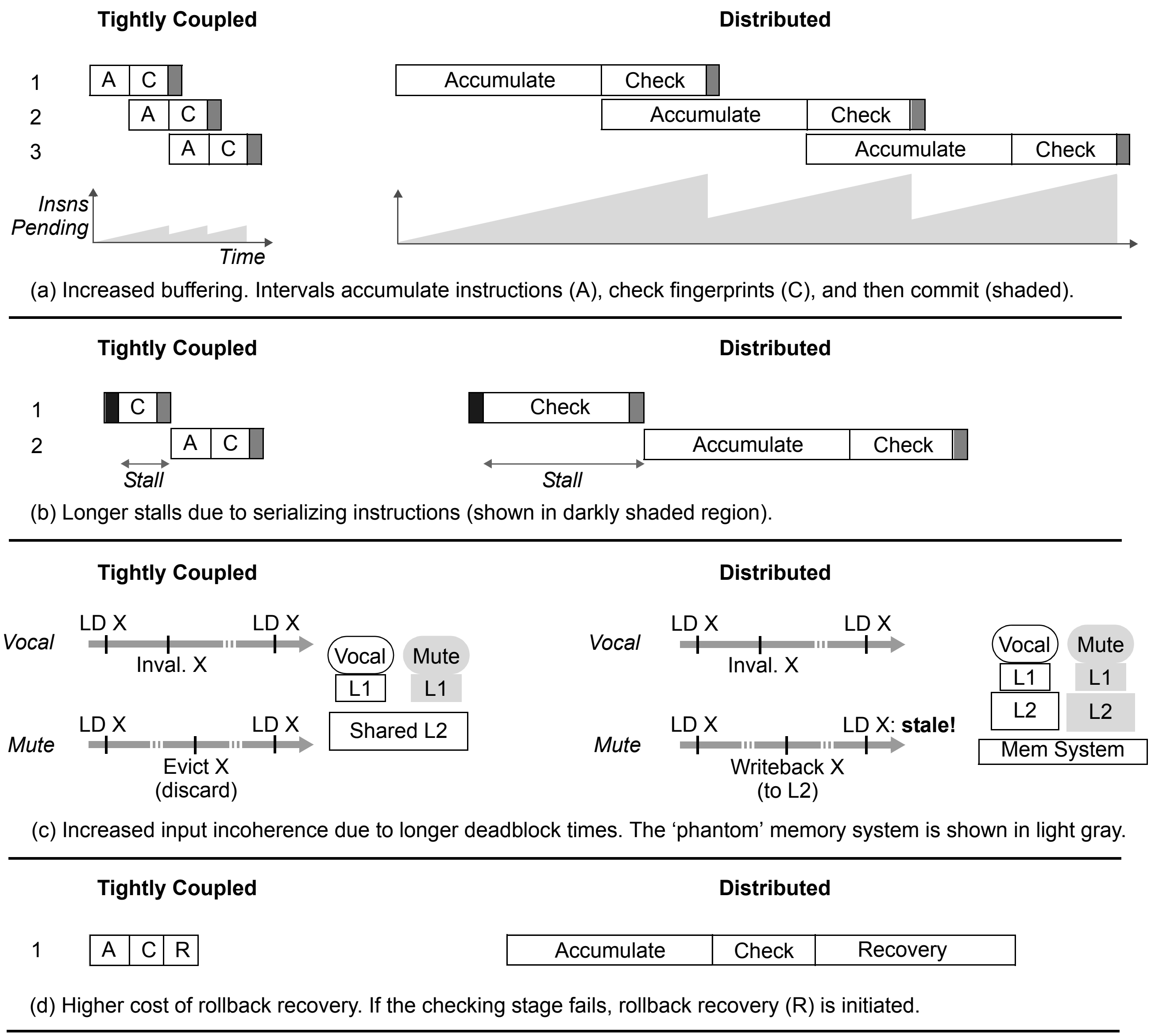
Figure 2 illustrates the key challenges that prevent a simple extension of the tightly coupled design in [37] from supporting cross-chip distributed redundancy. As in Reunion, we divide the dynamic instruction sequence into intervals (shown as a numbered sequence in the diagram) and compress status updates with fingerprints [38]. We illustrate the execution of a series of fingerprint intervals using a pipeline diagram, where each interval must first accumulate instructions as they retire from the processor core, then a fingerprint check is performed, and finally the instruction interval is committed as permanent changes to architectural state.

Distributing the redundant pairs reduces bandwidth available for checking fingerprints and increases the comparison latency, thereby increasing drastically the number of pending instructions that must be buffered (Figure 2(a)). The tightly coupled design proposed in [37] leveraged existing buffering resources within the out-of-order instruction window (e.g., 128 to 256 instructions). Our investigations show that naively extending this approach to a distributed design results in a 2 to 3X slowdown, as execution must continually stall while waiting for previous intervals to check and commit.

The second challenge comes from serializing operations, such as memory ordering instructions, instructions that access special, non-renamed registers (e.g., timers), and non-idempotent instructions (e.g., programmed I/O), which stall the pipeline until checking completes and the instruction is committed. As the checking latency increases, these operations become a larger performance bottleneck (Figure 2(b)). Of these, memory ordering instructions are by far the most common and, in the workloads we study, account for as much as a 40% performance overhead due to exposed checking latencies.

The replicas in the Reunion execution model, which are referred to as vocal and mute, access their local cache hierarchy as a non-redundant system would. To preserve existing coherence protocols, Reunion specifies that the mute portion of the memory system is *phantom*, meaning coherence state is not tracked and memory requests from the mute do not have to be coherent. As Figure 2(c) illustrates, because the mute's cache state is not tracked, it does not get invalidated when a new value is written to a location the mute has cached.

In a tightly coupled design, the mute's phantom memory system includes just the L1 cache and rollback recovery with a synchronizing request is comparable in latency to an L2 cache hit. The deadblock time of the L1 cache—the time from the last access of a block to its subsequent eviction—is sufficiently short that the mute hierarchy is likely to have discarded the now-stale block. If the stale block has not been evicted prior to the next use, the performance loss is minimal because of the short rollback recovery times. Neither of these conditions is true in the distributed case, as the deadblock times now include the much larger L2 or L3 cache and rollback discards more instructions and requires one or more off-chip memory accesses (Figure 2(d)). Our results indicate execution times double in several workloads due to these limitations.



(a) Increased buffering. Intervals accumulate instructions (A), check fingerprints (C), and then commit (shaded).

(b) Longer stalls due to serializing instructions (shown in darkly shaded region).

(c) Increased input incoherence due to longer deadblock times. The ‘phantom’ memory system is shown in light gray.

(d) Higher cost of rollback recovery. If the checking stage fails, rollback recovery (R) is initiated.

FIGURE 2. Performance impediments for distributed, asynchronous redundancy.

3 Mechanisms to Support Distributed, Asynchronous Redundancy

In this section, we present the *REDAC* mechanisms that enable distributed, asynchronous redundancy in shared-memory systems with little performance loss.

3.1 Scalable Result Buffering

We require a mechanism to buffer the results of a large number of instructions until a fingerprint check is performed. Several potential mechanisms have been proposed recently in the literature [2,8,12,18,25,43]. In essence, these mechanisms provide checkpoints of the register file at the beginning of a speculative window (our fingerprint interval) and then buffer store values either in a scalable structure [2,25], or obtain the effect of a scalable store buffer by leveraging capacity in the cache hierarchy [8,18,43].

Two constraints arise when designing a checkpoint and buffering scheme: the number of register checkpoints required and the total number of stores that need buffering. While one checkpoint is always required for the interval currently being accumulated, additional checkpoints are necessary to overlap the checking latency. Prior work [2,13,18] has developed mechanisms with support for a small number of register checkpoints (e.g., 4 simultaneous). Because of the long-latency speculation we require, checkpoints must include common instruction and floating-point registers as well as control and special-purpose registers. As in [43], we also record changes made to the TLB/MMU state to support speculation in supervisor code.

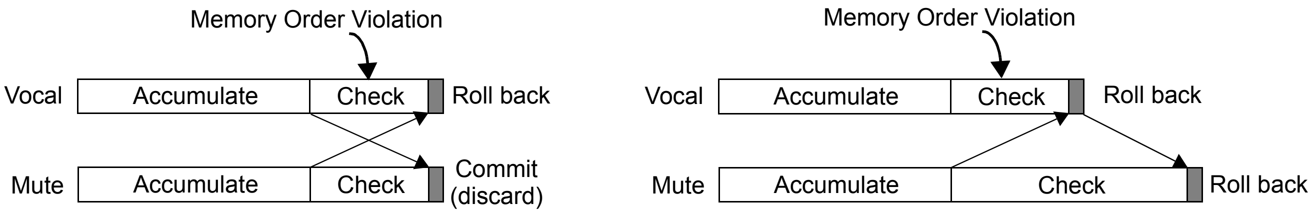


FIGURE 3. Fingerprint checking protocols. The left figure shows the fingerprint swap protocol proposed in [37], while the right shows the two-phase fingerprint check proposed in this work. If a memory order violation is detected during checking, the swap protocol may result in lost updates on the mute. The two-phase protocol synchronizes the rollback operation.

The store buffer serves a dual-purpose role by providing the most-recent value to the local processor and maintaining the specified memory order as stores are committed. A scalable buffering scheme decouples these two functions into a CAM structure for providing most-recent values to the processor and a FIFO for maintaining memory order. This is the approach taken in [43], which reuses the L1 cache as the CAM structure and constructs a separate FIFO on the side. The L1 cache tags are extended with per-word valid bits that permit the processor to read private values even though store permission has not been granted (the rest of the block is invalid); the per-word valid bits allow the block to be merged with new, local store values.

While mechanisms exist that enable large fingerprint intervals, we also require logic to decide when to end the current interval that the processor is accumulating. Because the DMR pair is asynchronous, the decision to end an interval cannot be based on external, timing-dependent events; rather, the only common reference is the instruction stream. Our design creates new fingerprint intervals under the following conditions:

- After an interval reaches a fixed, maximum number of instructions
- Before retiring an instruction with side-effects (e.g., programmed I/O)
- Memory operation following rollback
- Before and after retiring memory ordering instructions (e.g., atomics and FENCE/MEMBARs)

In Section 3.2, we show that creating a new interval on memory-ordering instructions is not strictly required if the hardware supports speculation on the memory model; however, synchronization code benefits substantially from shorter intervals (Section 3.3).

3.2 Speculative Serializing Instructions

We observe that mechanisms such as atomic sequence ordering (ASO) [43] that enable speculation beyond the retirement of the most-common serializing instructions—atomic memory accesses and memory ordering instructions—can also hide the long checking latency of distributed redundancy. Speculatively relaxing the memory model leads to possible rollbacks beyond conventional instruction retirement. Figure 3 illustrates the problem these post-retirement rollbacks pose for the fingerprint checking protocol of Reunion [37].

If the vocal sends a fingerprint, but later discovers an ordering violation (misspeculation), the mute may have already compared the vocal's fingerprint and advanced beyond the point of recovery. Although the vocal's checkpoint state could be copied to the mute to provide correct forward progress, this operation is costly and would cause significant performance degradation with frequent misspeculations.

Our approach, shown in Figure 3(right), converts the *swap* protocol from [37] into a *two-phase* checking process where the mute first sends its fingerprint, the vocal does a comparison after ensuring the sequence can commit, and then the vocal replies to the mute with its fingerprint. Although the checking latency for the mute is effectively doubled—placing additional pressure on the mute's buffering resources—the two-phase checking protocol permits the vocal to synchronously notify the mute when rollback is required due to ordering violations.

Implementation. As in ASO, we use a table of fingerprint intervals (atomic sequences in [43]) to track the status of blocks written speculatively by each interval. Each entry in the table has a count of the number of unique cache blocks written by the interval, as well as the number of cache blocks with writable permission. At vocal processors, intervals may commit when these two counts are equal and the fingerprint comparison has succeeded. Mute processors require only a fingerprint comparison before commit, as they are not required to be coherent or maintain memory consistency in the Reunion execution model.

To detect memory order violations, the L1 and L2 cache tags are augmented with a speculatively read bit for each pending sequence. Any read that follows a memory ordering instruction must mark its cache line with the speculative bit. If another processor invalidates a speculatively read line, the processor is notified of the ordering violation and must roll back.

Implementing rollback for memory ordering violations is challenging because the vocal detects memory order violations independent of fingerprint comparisons, so the mute

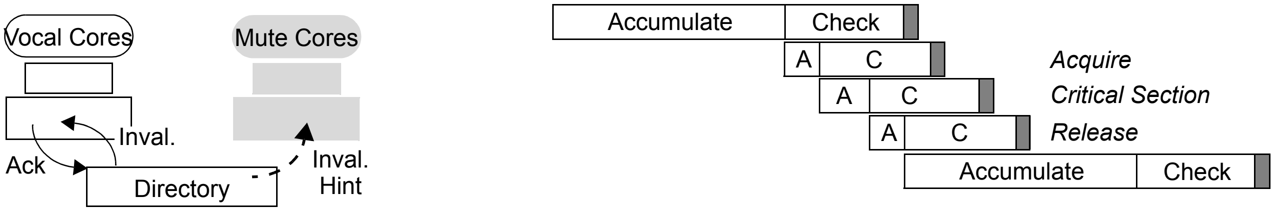


FIGURE 4. Limiting Input Incoherence. The left figure shows mute invalidation hints, and the right illustrates reduced fingerprint intervals around synchronization code.

may not have caught up to the recovery point. Our implementation stalls instruction retirement until the mute catches up—identified by fingerprint sequence numbers—then the vocal informs its mute of rollback by sending a poisoned fingerprint that causes the mute to roll back. In an aggressive implementation, the intervals older than the ordering violation can be kept; thus, while the vocal is stalled waiting for the mute to catch up, it must continue performing fingerprint comparisons for older intervals that will not be discarded.

Non-Speculative Serializing Events. Although memory ordering instructions constitute the vast majority of serializing operations and can thus be speculatively overlapped, other instructions and events cannot be speculated past. These include: handling external interrupts, non-idempotent operations such as device accesses, and reading from timer or performance counter registers.

When an interrupt arrives at the vocal, we delay its delivery until the vocal completes another fingerprint interval, at which point the fingerprint is tagged with the interrupt data and sent to the mute. The vocal now enters the interrupt handler. Because we use the two-phase checking protocol, the mute is guaranteed to have run ahead of the vocal at this point and must be rolled back before entering the interrupt handler.

For serializing instructions, no rollback is required because both vocal and mute can end the previous fingerprint interval immediately before the serializing instruction. The vocal then stalls until all prior intervals are checked and committed, and then sends the mute a fingerprint for the serializing instruction. In the case of non-renamed registers, the fingerprint packet must also include the register value to be used. We assume that these registers are hardened cells that mitigate the likelihood of a transient fault.

3.3 Limiting Impact of Input Incoherence

Frequency of Recovery. With the mute cache hierarchy including large L2 or L3 caches, deadblock times in our system will be much larger than in tightly coupled designs where the mute cache hierarchy is just the L1 cache [37]. In a directory-based coherence protocol, only sharers of a cache block get invalidated when a new value is written. Because the mute caches are not kept in the coherence-protocol state, the mute caches do not get invalidated and input incoherence may result on a later load instruction.

To limit the frequency of input incoherence due to these stale blocks, we add *mute invalidation hints* to the coherence protocol. As Figure 4(left) illustrates, when a vocal is invalidated during normal coherence protocol transitions, a hint is sent to its corresponding mute. These hints do not affect protocol transitions, nor do they require replies. These hints do not eliminate the possibility of input incoherence on real data races; rather, they eliminate non-races as a source of input incoherence.

Cost of Recovery. A distributed redundancy system has less bandwidth available for output comparison than a tightly coupled design, and hence the fingerprint intervals must include more instructions to amortize the cost of fingerprint traffic. As a result, the cost of recovery on input incoherence goes up considerably, as more instructions must be discarded upon rollback. We observe that, because input incoherence is now limited to data races, rollback will occur in and around synchronization code (e.g., critical sections protected by locks).

Modern processors provide hardware support for fast synchronization code with atomic memory operations and memory ordering instructions. As shown in Figure 4(right), by creating new fingerprint intervals before and after synchronization instructions, *REDAC* limits the window of vulnerability for input incoherence to actual races for shared variables. Because we support unmodified application and system code, *REDAC* must create new sequences around every atomic memory operation and ordering instruction. This handles complex synchronization where multiple locks are acquired before entering a critical section, a common pattern in system code.

Creating a new interval before the ordering operation reduces the cost of rollback by not discarding useful work prior to the ordering instruction, while a new interval immediately after the ordering operation reduces the frequency of rollback by limiting the time window where other processors to attempt to access the synchronization variable(s).

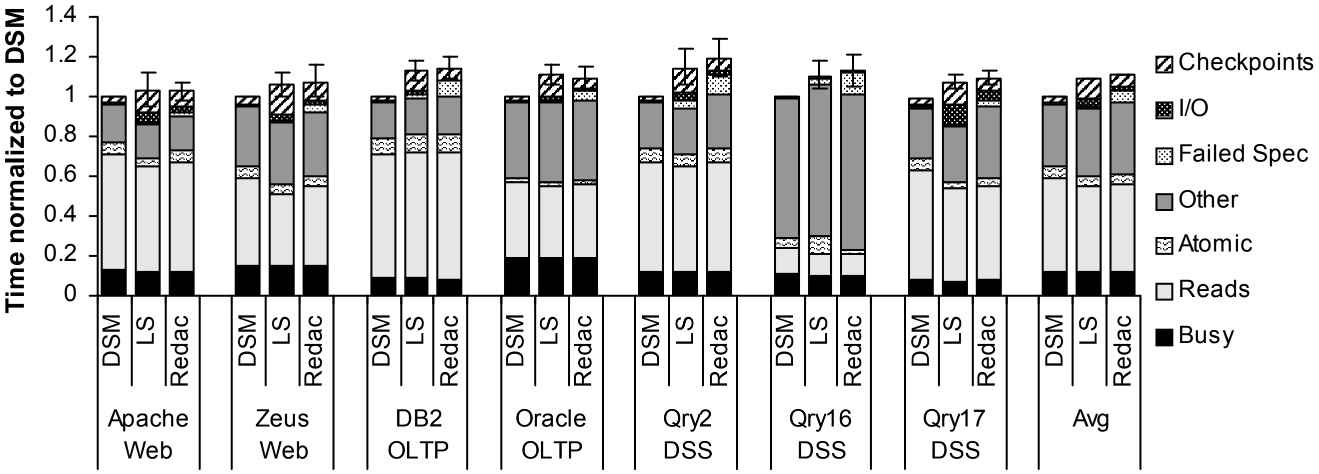


FIGURE 5. Baseline Performance Comparison. Each bar shows an execution-time breakdown for a non-redundant DSM, lockstep design, and *REDAC*, normalized to the non-redundant system. Error bars show 95% confidence intervals, and time is accounted as follows. 'Checkpoints' occurs when buffering resources are exhausted. 'I/O' is all interrupts and memory-mapped I/O instructions. 'Failed Spec' includes input incoherence (*REDAC* only) and memory model mis-speculation (all designs). 'Other' is primarily non-forward-progress instructions, such as lock spinning and the idle loop. 'Atomic' and 'Reads' are the read portion of the respective memory operations. 'Busy' is instruction retirement (non-stall cycles).

4 Evaluation

We evaluate *REDAC* using cycle-accurate full-system simulation in Flexus [44]. Flexus models the SPARC v9 ISA and can execute unmodified commercial applications and operating systems. Flexus extends the Virtutech Simics functional simulator with models of an out-of-order processor core, cache hierarchy, protocol controllers and interconnect. Our baseline processor includes support for memory model speculation using atomic sequence ordering (ASO) [43]. We simulate a 32-node system with 16 processor pairs in a directory-based shared-memory multiprocessor system running Solaris 8. We implement a directory-based NACK-free cache-coherence protocol, including added support for phantom memory requests and synchronizing requests. We list other relevant parameters in Table 1.

Table 2 enumerates our commercial application suite. We include the TPC-C v3.0 OLTP workload on IBM DB2 v8 ESE and Oracle 10g Enterprise Database Server. We run three queries from the TPC-H DSS workload on DB2, selected according to the categorization of Shao et al. [33]. We evaluate web server performance with the SPECweb99 benchmark on Apache HTTP Server v2.0 and Zeus Web Server v4.3. We drive the web servers using a separate client system (client activity is not included in results).

We measure performance using the SimFlex multiprocessor sampling methodology [44]. The SimFlex methodology extends the Smarts [45] statistical sampling framework to multiprocessor simulation. Our samples are drawn over an interval of from 10s to 30s of simulated time for OLTP and web server applications and over the complete query execution for DSS. We launch measurements from checkpoints with warmed caches and branch predictors, then warm queue and interconnect state for 100,000 cycles prior to measuring 50,000 cycles. We use the aggregate number of user instructions committed per cycle (i.e., committed user instructions summed over the 16 processor pairs divided by total elapsed cycles) as our performance metric, which is proportional to overall system throughput [44]. We do not include error events in our evaluation; however, input incoherence events, output comparison, and recovery are modeled in detail.

4.1 *REDAC* Overview

We evaluate the performance of *REDAC* by comparing with a non-redundant architecture ("DSM") and a redundant design that uses lockstep replication ("LS"). The lockstep

TABLE 1. System parameters.

Processing Nodes	UltraSPARC III ISA 4 GHz 8-stage pipeline; out-of-order 4-wide dispatch / retirement 128-entry ROB, LSQ 32-entry store buffer
L1 Caches	Split I/D, 64KB 2-way, 2-cycle load-to-use 4 ports, 32 MSHRs, 16-entry victim cache
L2 Cache	Unified, 8MB 8-way, 25-cycle hit latency 1 port, 32 MSHRs
Main Memory	3 GB total memory 60 ns access latency 64 banks per node 64-byte coherence unit
Protocol Controller	1 GHz microcoded controller 64 transaction contexts
Interconnect	4x4x2 torus 25 ns latency per hop 256 GB/s peak bisection bandwidth

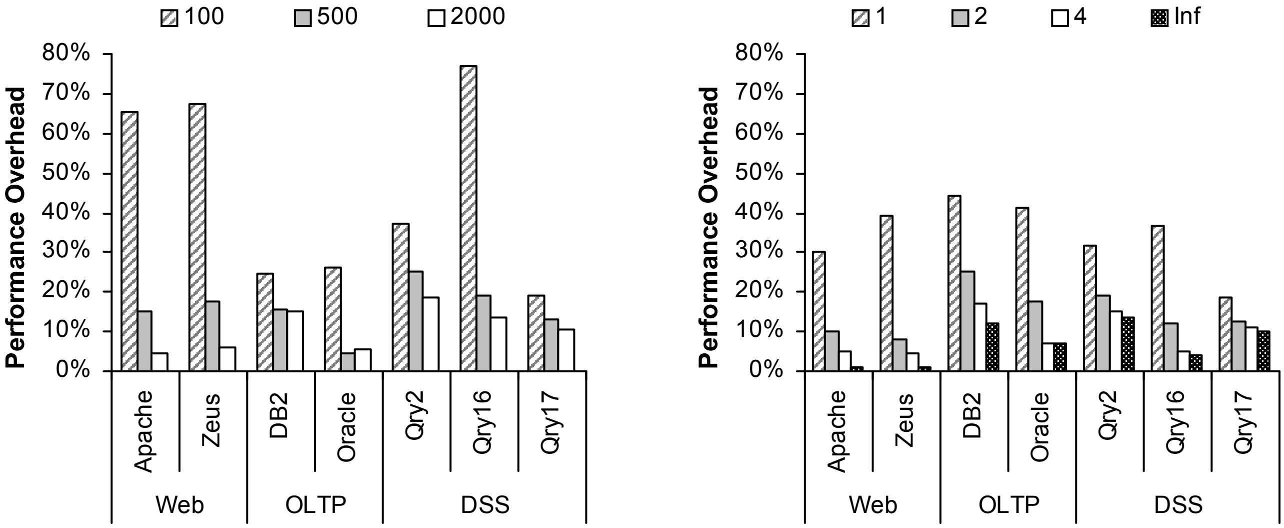


FIGURE 6. Buffering Requirements. The left figure shows sensitivity to the nominal fingerprint interval. The right figure shows sensitivity to the number of available register checkpoints, where 'Inf' allows an unbounded number. All results are normalized to the non-redundant DSM, which has four register checkpoints.

architecture is configured identically to the *REDAC* system, except instead of using asynchronous redundancy, our lockstep model replicates all chip-external inputs as in the TRUSS [15] design. Our lockstep approach differs from TRUSS in how we handle 'dirty read' operations, which occur when a processor reads a value modified previously by a remote processor. Rather than delay the reader to check that the updated cache line is free of errors, we leverage the scalable store buffering provided by our baseline microarchitecture to hide unchecked stores. As in the *REDAC* system, writebacks from the mute cache are discarded under the lockstep design.

Figure 5 shows the execution-time breakdown for these models, normalized to the non-redundant DSM. The redundant designs both use nominal fingerprint intervals of 2000 instructions, and all three systems support up to four register checkpoints. *REDAC* achieves similar overheads (10%) as the lockstep design, without requiring determinism or full-state initialization. Despite the similar overall performance, *REDAC* and lockstep spend the additional stalls differently. The lockstep design has a fixed slip—the absolute time difference between vocal and mute—that corresponds to the worst-case, single-hop network latency. For our interconnect this is 140 ns (560 processor clock cycles). The relaxed input replication of *REDAC* permits a variable slip that averages 25 ns (the minimum network-hop latency). This difference manifests in the higher 'Checkpoints' and 'I/O'-related stalls for the lockstep design, as both expose the slip on the critical path of execution. From the *REDAC* perspective, these improvements are offset by a higher rate and cost of failed speculation, primarily from input incoherence.

TABLE 2. Application parameters.

Online Transaction Processing (TPC-C)	
DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web Server	
Apache	16K connections, fastCGI, worker threading model
Zeus	16K connections, fastCGI
Decision Support (TPC-H on DB2)	
Qry 2	Join-dominated, 450 MB buffer pool
Qry 16	Join-dominated, 450 MB buffer pool
Qry 17	Scan/join-balanced, 450 MB buffer pool

4.2 Buffering Requirements

Our baseline performance analysis shows that on average *REDAC* performs within 10% of a non-redundant system. In Figure 6, we examine the sensitivity of *REDAC* to key design choices that affect buffering requirements: nominal fingerprint interval and the maximum number of register checkpoints. Unlike the number of checkpoints, the fingerprint interval size has no associated hardware cost. However, as Figure 6(left) shows, there is substantial performance sensitivity to this parameter choice. With shorter intervals, fingerprints are exchanged more frequently, and network bandwidth constraints limit performance. Because we found little sensitivity beyond 1000 instructions, we chose 2000 as the baseline parameter.

In Figure 6(right), we show the sensitivity to the number of register checkpoints available for speculation. We observe that execution time with four register checkpoints is within 5% of the unbounded case across all the applications we studied, while having just two checkpoints loses an additional 10% in the worst case (Oracle OLTP). Having a single checkpoint exposes a portion of the fingerprint comparison latency any time the checking latency is larger

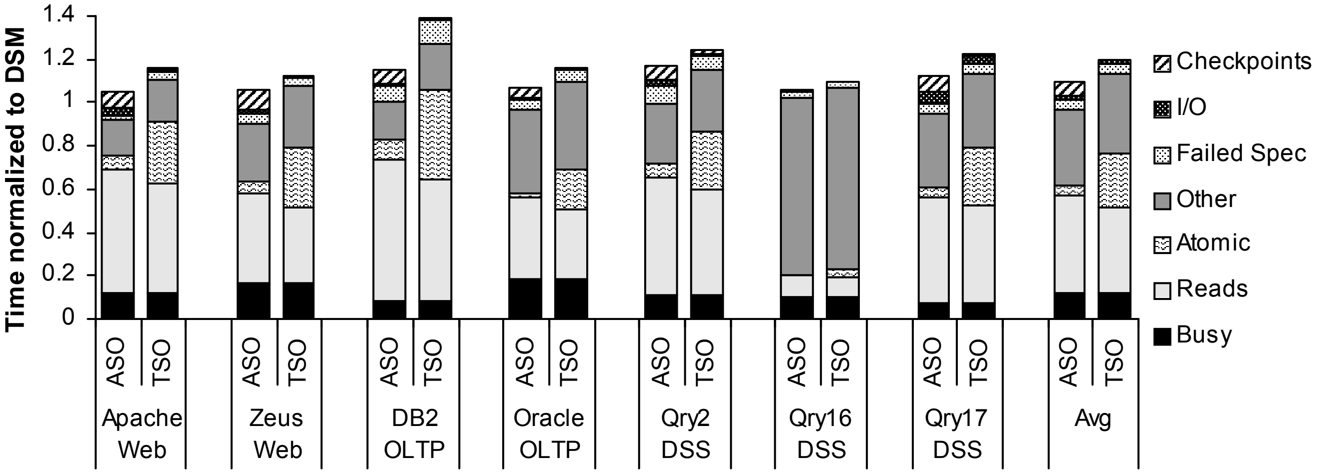


FIGURE 7. Impact of Memory Model Speculation. A distributed redundancy scheme benefits substantially from speculation on the memory order (ASO). Abiding by the SPARC TSO model means even uncontended locks that hit in cache must stall for cross-chip fingerprint comparisons.

than the time until the processor's instruction window is exhausted. In our model, checking latencies average between 100 and 200 processor clock cycles (25 to 50 ns), and our 128 entry ROB will fill in 64 to 256 cycles for the applications we studied, on average. Thus, as Figure 6(right) shows, significant stalls should be expected.

For comparison, we examined a variant of *REDAC* that used no register checkpoints, but instead relied only on speculation within the out-of-order instruction window. On average, execution time more than doubles, as the instruction window lacks sufficient buffering to overlap the checking latency.

4.3 Speculating on Ordering Instructions

Figure 7 shows the impact of ordering instructions—atomic memory operations and memory barriers—if, despite having the necessary hardware for memory order speculation, *REDAC* was limited to implementing the Total Store Order (TSO) model specified by the SPARC architecture. In such a scenario, the processor must stall while all older store operations are committed, including all prior checkpoints still pending fingerprint comparisons. This is particularly problematic for atomic operations that hit in the cache, because rather than complete in a few cycles, they must stall for a significantly longer fingerprint operation.

4.4 Limiting Input Incoherence

Because of the latencies involved in distributed redundancy, rollback has a high performance cost. Our experiments show that, on average, rollback takes approximately the same amount of time as one to two off-chip memory accesses (800 processor cycles in our model). Thus, it is of critical importance to minimize the number of input incoherence events. Figure 8 illustrates two approaches to meeting this goal: mute invalidation hints and generating new intervals on ordering instructions.

Because the mute is not required to be a coherent part of the memory system, it is permitted to hold stale cache blocks even though another DMR pair has updated the value. Allowing the mute nodes to be stale is particularly problematic in *REDAC* because the large L2 cache in the mute nodes often holds cache blocks for millions of cycles or longer. If the mute is not informed of invalidations by other DMR pairs, it will very likely read stale values and require rollback due to the input incoherence. Figure 8(left) confirms this hypothesis, indicating an average performance overhead of 30%. In DB2 OLTP, the overhead is nearly 75% because of migratory data patterns.

Table 3 quantifies the frequency of input incoherence both in our baseline design for *REDAC* and when mute invalidation hints are omitted. Compared to the values reported from Reunion [37], *REDAC* shows a 10-100X increase in input incoherence frequency. Omitting the mute invalidation hints increases the rollbacks by another order of magnitude. We observe two potential reasons for why input incoherence is more frequent in *REDAC* than in a tightly coupled design like Reunion. First, the applications we study in this paper are scaled to 16 logical processors, and lock contention increases accordingly. Our investigations suggest

Table 3. Input incoherence rates, per 1M instructions.

Workload	With Mute Invalidations	No Mute Invalidations
Apache	11.2	86.5
Zeus	9.4	83.2
DB2 OLTP	46.7	491.5
Oracle OLTP	18.7	140.2
DB2 DSS Q2	31.0	152.0
DB2 DSS Q16	28.8	139.6
DB2 DSS Q17	30.0	214.7

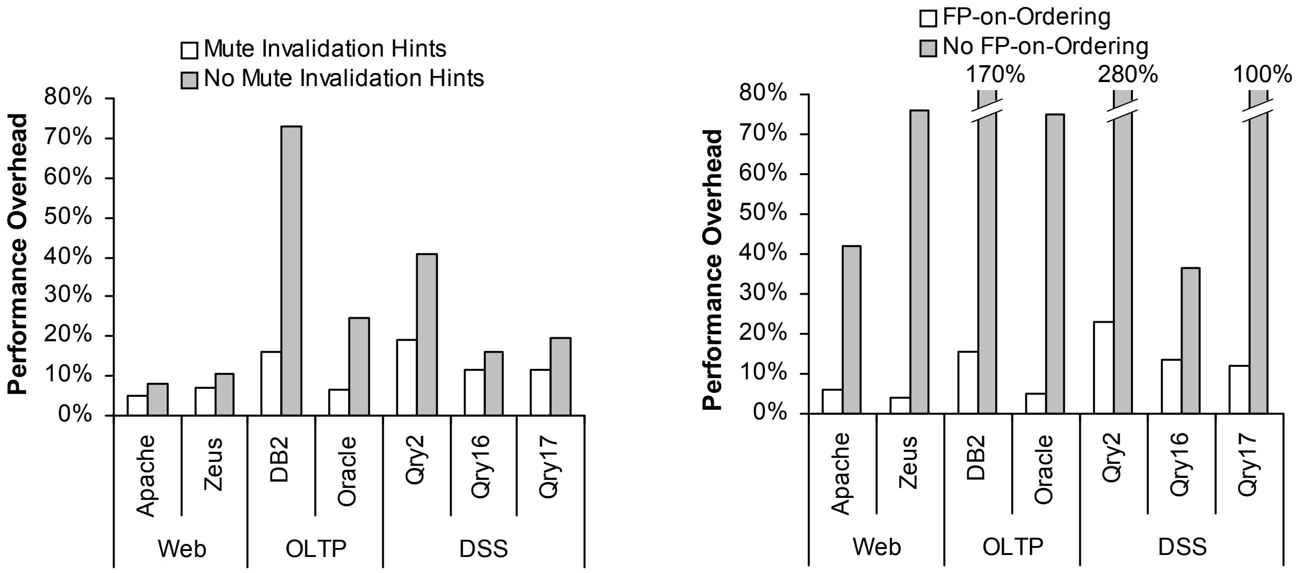


FIGURE 8. Limiting Input Incoherence. The left figure shows the necessity of providing mute nodes invalidation hints as part of the coherence protocol. The right figure shows the impact of creating short fingerprint intervals around memory ordering and atomic operations.

that lock contention accounts for nearly all input incoherence in these applications, with some additional rollbacks attributable to false sharing, particularly in the TCP stack in Solaris 8. Second, we note that the slip between vocal and mute is likely to be more variable in *REDAC* than in Reunion, as the cross-chip interconnect and memory-system components have far more sources of variable queuing delays than a shared on-chip cache. Although the median slip is a single network hop—25 ns in our design—we observed slips as large as a few microseconds. A slip of this magnitude corresponds to a large window of vulnerability for input incoherence.

Additionally, we note the importance of isolating memory ordering instructions into short fingerprint intervals with Figure 8(right). In this experiment, we do not generate new fingerprint intervals before *or* after retiring a memory ordering instruction. We serialize these instructions at significant cost; however, they are rare. The large performance overheads are due to memory order violations, which require rollback with cost similar to input incoherence. The increase in memory order violations results from holding locks—they are privately released but publicly still held—well beyond when the critical section ends, thus increasing contention for shared data. It is vital to commit lock releases as soon as possible.

4.5 Slipstream Effects

To minimize the slip between vocal and mute, we proposed that *REDAC* distribute physical memory and accompanying directory entries across all the nodes. And while we observed a substantial increase in slip for the worst case, the median slip was largely unchanged if we placed the memory and directory entries only on vocal nodes. As Figure 9 shows, the performance is very similar to the baseline. We include this result because it showed an interesting phenomenon: slipstreaming [29]. On average, vocal requests for remote cache blocks will arrive at the directory controller first in this scenario, with the mute request arriving 25 to 50 ns later on average. The directory state is stored in DRAM, along with the actual data, and both take 60ns to access. The mute's request will hit in the cache in the directory, eliding nearly all of the DRAM latency. Thus, although the mute nodes have an additional network hop to traverse for every memory access, performance does not suffer.

5 Related Work

The conventional approach to redundant computation uses clock-for-clock lockstep [34], where replicas are tied to the same physical clock, and inputs (e.g., cache line fills and interrupts) and outputs (e.g., cache writebacks and programmed I/O operations) are synchronized to the common clock. Implementing any lockstep redundancy scheme requires initialization of all state-holding elements—including those that do not affect architecturally correct execution (e.g., branch predictors [24])—and completely deterministic execution.

Because of these challenges, current mainframes from HP [5] and IBM [42] have moved away from synchronous redundancy. Instead, HP's NonStop Advanced Architecture (NSAA) synchronizes inputs and outputs at software-defined boundaries. Aggarwal et al. [1] extend the NSAA architecture to address common-mode failures within commodity CMPs; however, the NSAA approach requires custom system and application software. After several generations of tightly coupled DMR pipelines [35], IBM

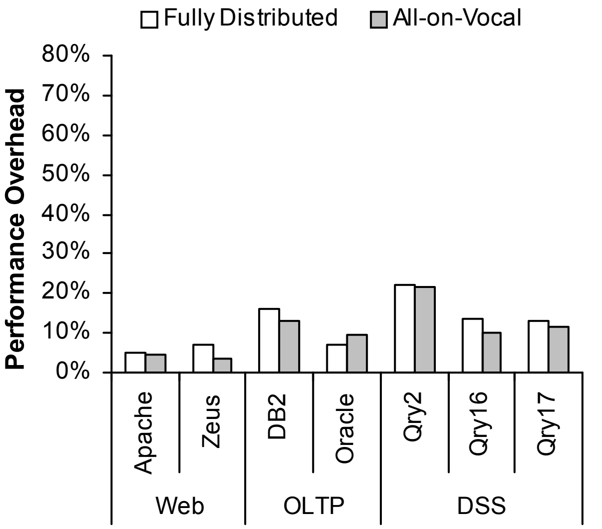


FIGURE 9. Memory Placement. Although slip increases slightly, unbalancing the system by placing memory only on vocal nodes does not hurt performance because the mute nodes can slipstream vocal memory requests.

has moved away from DMR altogether, instead opting to embed over 20,000 fine-grained checkers into their custom z-Series chips [42]. As in previous-generation designs, IBM's mainframes rely on extensive software support for high availability.

Our approach to checkpointing differs from recent work in that we advocate fine-grained checkpointing at the granularity of cache coherence interactions. In ReVive [28] and SafetyNet [39], checkpoints are created on the order of milli- and micro-seconds. Coarse-grained checkpointing improves system availability, but requires kernel extensions and specific network protocol features to tolerate I/O intensive workloads [26].

Recent work has also addressed the emergence of hard errors in future processor designs. Srinivasan et al. [40] applies device failure models to predict future hard errors and proposes techniques to extend the operational lifetime, but cannot detect in-field faults when they occur. Bower et al. [7] use a checker core to detect hard- and soft-errors at runtime, but requires a custom checker core to do so. The BulletProof design [10] integrates custom built-in self test (BIST) hardware into the processor pipeline, sending test vectors through periodically to detect recent hard faults.

Montesinos [21] observes that breaking instructions into intervals (referred to as chunks in [21]) and leveraging memory order speculation can aid logging and deterministic replay of multiprocessor programs. Although deterministic replay shares many commonalities with redundant execution, the key difference is that redundant processors in *REDAC* must wait for the checking operation to complete before intervals commit, whereas deterministic replay need only log the order of interval commits.

6 Conclusion

We proposed *REDAC*, a set of lightweight mechanisms for distributed, asynchronous redundancy within a scalable shared-memory multiprocessor. *REDAC* provides scalable buffering for unchecked state updates and overcomes common performance bottlenecks from previous designs, permitting the distribution of redundant execution across multiple nodes of the DSM server. With asynchronous redundancy, *REDAC* obviates the extensive initialization and determinism requirements of prior lockstep-based designs. We evaluated *REDAC* using cycle-accurate full-system simulation of common enterprise workloads and showed that performance overheads average just 10% when compared to a non-redundant system. These results are comparable to the performance of a similarly configured lockstep design, but offer the substantial benefits of asynchronous redundancy.

Acknowledgements

The authors would like to thank the members of the TRUSS and Impetus research groups at CMU for their feedback on this paper. This work is supported by NSF CAREER award CCF-0347568, NSF award ACI-0325802, Sloan and DoD/NDSEG fellowships, the Center for Circuit and System Solutions (C2S2), MARCO, CyLab, and by grants and equipment from Intel Corporation.

References

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: Building high availability systems with commodity multi-core processors. In *Proc. 34th Intl. Symp. Computer Architecture*, 2007.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, 2003.

[3] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, 1999.

[4] W. Bartlett and B. Ball. Tandem's approach to fault tolerance. *Tandem Systems Rev.*, 8:84–95, Feb 1988.

[5] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *Proc. Intl. Conf. Dependable Systems and Networks*, 2005.

[6] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–17, Nov-Dec 2005.

[7] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc 38th Intl. Symp. on Microarchitecture (MICRO 38)*, Dec 2005.

[8] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Transactions on Architecture and Code Optimization*, 3(2):182–208, 2006.

[9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proc. 34th Intl. Symp. Computer Architecture*, 2007.

[10] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: A defect-tolerant CMP switch architecture. In *Proc. 12th Intl. Symp. on High-Performance Computer Architecture*, 2006.

[11] Intel Corporation. Intel QuickPath architecture. *Intel Whitepaper*, 2008.

[12] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Intl Symp on High Performance Computer Architecture*, 2004.

[13] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *Proc. of ICCD*, 2004.

[14] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Intl. Symp. on Computer Architecture*, May 1999.

[15] B. T. Gold, J. Kim, J. C. Smolens, E. S. Chung, V. Liaskovitis, E. Nuvitadhi, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. TRUSS: a reliable, scalable server architecture. *IEEE Micro*, 25:51–59, Nov-Dec 2005.

[16] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. Int'l Symp. on Computer Architecture,* 2003.

[17] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD opteron processor for multiprocessor servers. *IEEE Micro*, March-April 2003.

[18] J. F. Martinez, J. Renau, M. . C. Huang, and M. Prvulovic. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proc. of the 35th IEEE/ACM Intl. Symp. on Microarchitecture*, Nov 2002.

[19] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower. IBM z990 soft error detection and recovery. *IEEE Trans. device and materials reliability*, 5(3):419–427, Sept 2005.

[20] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures: The Out of Order Reliable Superscalar O3RS approach. In *Proc. of the Intl. Conference on Dependable Systems and Networks*, June 2000.

[21] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proc. 35th Intl. Symp. Computer Architecture*, 2008.

[22] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: an architectural perspective. In *Proc. Intl. Symp. on High-Performance Computer Architecture*, 2005.

[23] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. Int'l Symp. Computer Architecture*, May 2002.

[24] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. 36th IEEE/ACM Intl. Symp. on Microarchitecture*, 2003.

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, November/December 2003.

[26] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *Int'l Symp. High-Performance Computer Architecture (HPCA)*, Feb 2006.

[27] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. 7th ACM Intl. Conference on Management of Data*, 1988.

[28] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared memory multiprocessors. In *Proc. 29th Intl. Symp. on Computer Architecture*, 2002.

[29] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proc. of the 33rd Intl. Symp. on Microarchitecture*, 2000.

[30] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proc. of the 34th Intl. Symp. on Microarchitecture*, 2001.

[31] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. Int'l Symp. Computer Architecture*, 2000.

[32] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proc. 29th Intl. Symp. on Fault-Tolerant Computing*, June 1999.

[33] M. Shao, A. Ailamaki, and B. Falsafi. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Proc. 15th IBM Center for Advanced Studies Conference*, 2005.

[34] D. P. Siewiorek and R. S. Swarz. (Eds.). *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[35] T. Slegel, R. A. III, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[36] J. C. Smolens. *Fingerprinting: Hash-Based Error Detection in Microprocessors*. PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, Jan. 2008.

[37] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc 39th Intl Symp. on Microarchitecture,* 2006.

[38] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proc.11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.

[39] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. 29th Intl. Symp. on Computer Architecture*, 2002.

[40] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proc. 31st Intl. Symp. Computer Architecture*, 2004.

[41] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. Intl. Conf. on Dependable Systems and Networks*, 2004.

[42] C. Webb. IBM z6 - the next-generation mainframe microprocessor. In *HotChips 19*, 2007.

[43] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proc. 34th Intl. Symp. Computer Architecture*, 2007.

[44] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[45] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation through rigorous statistical sampling. In *Proc. 30th Intl. Symp. on Computer Architecture*, June 2003.