

Tolerating Processor Failures in a Distributed Shared-Memory Multiprocessor

Brian T. Gold, Babak Falsafi, and James C. Hoe
 Computer Architecture Laboratory (CALCM)
 Carnegie Mellon University
<http://www.ece.cmu.edu/~truss>

Abstract

Scaling transistor geometries and increasing levels of integration lead to rising transient- and permanent-fault rates. Future server platforms must combine reliable computation with cost and performance scalability, without sacrificing application portability. Processor reliability—for both transient and permanent faults—represents the most challenging aspect of designing reliable, available servers.

In this paper, we investigate redundant operation of physically-distributed processors in the context of a distributed shared-memory (DSM) multiprocessor. Our design—LACROSS—addresses two key challenges, namely the coordination and the corroboration of decoupled, redundant processors. Evaluations based on a range of commercial and scientific workloads show that the average error-free performance overhead of LACROSS is within 5.8% (worst case 13%) of a baseline system.

1 INTRODUCTION

Many of the day-to-day digital services that we now take for granted, from accounting and commercial transactions to residential utilities, rely on available and reliable information processing and storage. Server reliability is already a key pillar for e-commerce where downtime directly impacts revenues and can cost greater than \$6M per hour for availability-critical services [30]. With the proliferation of digital services, reliability will become a key design metric for future server platforms.

While server availability and reliability are becoming crucial, there are a number of obstacles to designing, manufacturing, testing and marketing reliable server platforms [22,30]. One key obstacle is the drastic reduction in reliability in future gigascale CMOS technologies and circuits [7]. Not only will these circuits exhibit high transient error rates [27] (due to particle radiation), but also they will incur prohibitive rates of intermittent or permanent failure due to a myriad of factors including (but not limited to) manufacturing difficulties at gigascale level [8], thermally-induced or time-dependent transistor and interconnect degradation [8,38], and ineffectiveness of “burn-in” to mitigate chip infant mortality [40]. Moreover, permanent failures can affect transistors, circuits, or entire chips—requiring solutions at the system level to achieve acceptable server reliability and availability.

Unfortunately, current solutions to server reliability suffer from a number of shortcomings. Tightly-coupled shared-memory mainframes—such as IBM’s z-series [34]—only recover

from transient errors and intra-chip failure. As such, these designs preclude transparent recovery from a permanent chip failure, which will be needed in future gigascale integrated systems. Moreover, the use of custom processor cores and shared backplanes limit both cost and performance scalability in such designs.

In contrast, distributed memory solutions—such as HP/Tandem NonStop [6]—tolerate permanent processor chip failures through redundant execution across chips. Moreover, these designs rely on commodity processor chips and multistage interconnects and as such offer both cost and performance scalability. Unfortunately, they also sacrifice software transparency by requiring a message-passing programming interface and as such preclude running conventional shared-memory multiprocessor workloads.

In this paper, we investigate processor fault-tolerance in server architectures based on cache-coherent distributed shared memory (DSM). DSM is an attractive architecture because it offers a software-compatible scalability path beyond today’s dominant commercial server architectures—chip and symmetric multiprocessors. By relying on commodity processor cores and multistage interconnects as building blocks, DSM offers an inherently cost- and performance-scalable design point. Highly-integrated DSMs [11] also enable high-bandwidth and low-latency paths to both DRAM memory and neighboring nodes, further reducing system-level “glue” logic and improving complexity, cost and performance.

We make the observation that DSM is a natural platform for “node-level” dual-modular redundancy (DMR), where two physically-distributed processors execute an identical instruction stream. Unlike conventional DMR approaches that rely on tightly-coupled lockstep, node-level redundancy faces two key challenges: the DMR processor pair must (1) observe identical external inputs to maintain a redundant instruction stream; and (2) corroborate results under the limited bandwidth and non-negligible latency of the interconnect.

We address these challenges with *Lockstep Across System (LACROSS)*, a set of hardware mechanisms extending a baseline DSM to pair processors on different DSM nodes for dual modular redundant execution. LACROSS relies on a previously-proposed lightweight state comparison mechanism [35] and checkpointing across DMR pairs to detect and recover from transient error in processor cores. Node-level redundancy enables LACROSS to survive not only circuit- and microarchitecture-level failure, but also the complete loss of a processor chip.

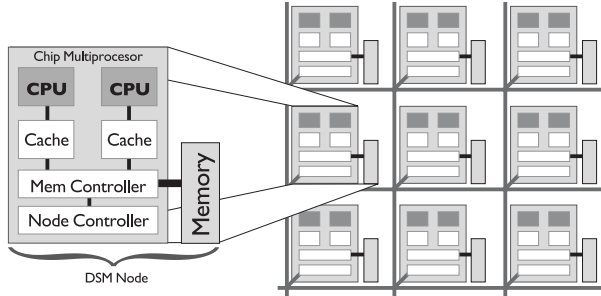


FIGURE 1: Hardware DSM.

We evaluate LACROSS with cycle-accurate full-system simulation of commercial and scientific workloads. The three main contributions of this paper are:

- **Lockstep coordination** — hardware mechanisms and protocols that maintain lockstep execution across decoupled processing modules separated by a scalable interconnect.
- **Error detection and recovery** — support for DMR error detection and recovery despite bandwidth and latency constraints of the interconnect.
- **Early-release performance optimizations** — a set of optimizations that significantly reduce the performance overhead induced by distributing redundancy. Our study shows that this performance overhead can be reduced to less than 5.8% on average (13% in the worst case) for a range of commercial and scientific workloads.

Paper Outline. This paper is organized as follows. In the next section, we provide background on distributed DMR in a DSM and elaborate on the key obstacles. We present the details of LACROSS coordination in Section 3 and describe error detection and recovery protocols in Section 4. Section 5 presents optimizations to remove common-case performance overheads. We evaluate the system’s performance in Section 6 using cycle-accurate, full-system simulation. Section 7 presents related work, and we conclude in Section 8.

2 BACKGROUND

Figure 1 depicts the anatomy of the cache-coherent DSM multiprocessors we study in this paper. Each node consists of a number of processor cores with their cache hierarchies, a memory controller scheduling accesses to local DRAM memory, and a node controller implementing a full-map directory cache-coherence protocol and the network interface and router logic to extend the local shared-memory abstraction across DSM nodes [26]. In a DSM, the inherent mirroring of physical resources across system nodes provides a natural platform for redundant execution. In this paper, we assume the nodes to be dual-core chip-multiprocessors (CMPs), although the ideas presented here extend to an arbitrary number of cores per node.

2.1 Membrane: Isolating Faults in DSM

To reason about faults in DSM servers, recent work introduced Membrane [18], a decomposition framework that abstracts the

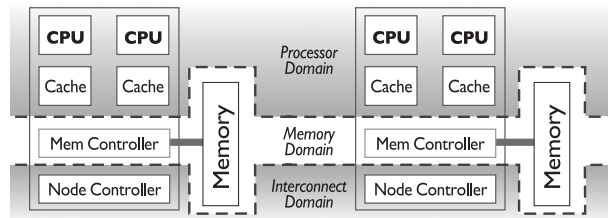


FIGURE 2: Membrane decomposition of a DSM. A fourth domain—devices—is not shown.

components in a large-scale server into *protection domains*. Membrane requires that errors occurring within a protection domain are detected *and corrected* before propagating to the rest of the system.

Figure 2 illustrates the Membrane decomposition used in this paper (excluding peripheral devices). Errors in instruction execution must be detected and corrected before propagating to DRAM. Cache-coherent data transfers from one processor to another must cross from the processor domain into the memory and interconnect domains. Because Membrane requires error detection and correction at each domain boundary, errors cannot propagate from one processor to another.

Membrane does not specify nor provide the error detection and correction mechanisms. Rather, the key enabler in this framework is that each domain can employ an optimized error detection and local recovery strategy. Although the CMP contains components from three Membrane protection domains, failure of the entire chip is orthogonal to where the Membrane boundaries are drawn. The fundamental requirement remains: components must detect and recover from errors within their domain boundary.

In this paper, we address failures in the processor domain, both transient and permanent. Recent work proposed parity-based error detection and recovery for soft errors in memory [18,32], and industry uses various techniques [10,12] to protect from permanent failures in DRAM and related circuitry. Membrane-compatible techniques to protect other elements of the DSM (e.g., interconnect [13,37], I/O devices [31], and node controllers [41]) are beyond the scope of this paper.

Fault Model. Our fault model addresses the vulnerability in existing processor designs for both transient and permanent faults. We assume single event upsets, due to cosmic rays and alpha particles, can manifest in soft errors that disrupt execution but do not permanently damage the hardware. Permanent faults, however, can damage any portion of the chip, including single transistors, circuits, or the entire chip itself.

In the case of soft errors, we assume the processor pipeline is vulnerable from fetch to retirement. Retired machine state—architectural register values and on-chip caches—are protected through information redundancy (e.g., ECC). With respect to permanent failure, we assume only one chip fails at a time.

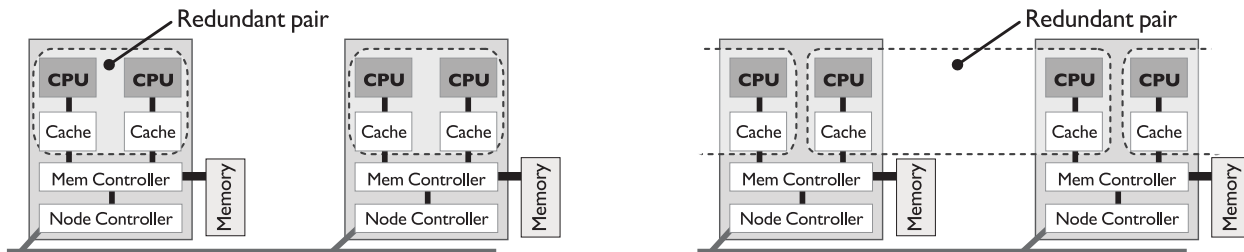


FIGURE 3: Organizing DMR in a DSM: (left) tightly-coupled pairs, (right) distributed pairs.

2.2 DMR and Distributed DMR

Reliable mainframes employ dual-modular redundancy (DMR) at various levels, ranging from replicating pipelines within the same die [34] to mirroring complete processors on a single system board [6]. In a conventional DMR design, computation is performed redundantly by duplicated hardware operating in lockstep.

To ensure the correctness of the processing module in a DSM, one could envision modifications pairing the two cores of a chip-multiprocessor for redundant computation [19,28]. Figure 3(left) illustrates such an organization. The DMR core pairs, driven by a common clock and bus, would be presented with external inputs—cache line fills, coherence requests, interrupts, etc.—at precisely the same time. Provided the two cores making up the DMR pair are deterministic in operation, the correct external outputs generated by the two cores must mirror each other exactly in value and timing. Thus, a comparison circuit can check the output of the two cores to detect an erroneous output (mismatch in value or timing). Provided an error is detected soon enough, the execution can be recovered to an earlier, known-good state and restarted.

However, given the goal of surviving a node failure, the cores constituting a DMR pair must reside on different nodes. Figure 3(right) suggests a possible distributed-DMR pairing of processing cores from different nodes. By distributing the redundant pair across nodes in the system, we are faced with two new fundamental problems:

- (1) we cannot guarantee simultaneous delivery of external inputs to both cores of the DMR pair, and
- (2) the two cores can only exchange information at a latency and bandwidth dictated by the interconnect.

In Section 3, we present an asymmetrical lockstep design that obviates the need for simultaneous delivery. In Sections 4 and 5, we present an error-detection technique and associated optimizations that minimize the performance overhead of distributed DMR despite the substantial communication cost between the partner cores.

3 COORDINATION IN LACROSS

When the DMR modules are situated on the same specially-designed chip or motherboard, DMR modules can achieve lockstep operation by sharing a common clock and using a common bus to receive input stimuli in precise synchrony. However, this direct approach to enforce lockstep is infeasible

in a distributed DMR arrangement. Instead, our solution, LACROSS, takes an asymmetrical approach to lockstepping where true simultaneity is not needed.

For the sake of discussion, first assume the DSM nodes operate on a globally synchronous clock. (In Section 3.1, we explain how this assumption can be relaxed.) The node controller on each node maintains a local timestamp for each core of the processing module. We require the execution of one core (the *slave*) in a DMR pair be delayed by a fixed lag relative to the other core (the *master*). In other words, timestamps of the master and slave advance synchronously but differ by a constant amount.

External inputs (including all incoming cache coherence activity, cache line refills, uncached load values, asynchronous inputs and interrupts) are only directed toward the leading master of a DMR pair, which appears as a single, logical processor to the system. These inputs can arrive at the master freely without extraneous requirements on delivery time. The delivery time of the external input at the master is recorded and forwarded together with the input value to the slave’s node controller in a *coordination message*. Provided the coordination messages always arrive early enough (i.e., before the slave core reaches the necessary delivery time), the perception of lockstep between master and slave is achieved by delivering the forwarded input to the slave at the correct corresponding local delivery time (illustrated in Figure 4).

The fixed master-to-slave delay must be sufficiently long to permit worst-case transit time of the coordination message between the master and slave in a distributed DMR pair. To bound worst-case transit time, we require coordination messages to travel on the highest-priority channel of the interconnect (cannot be blocked by other message types of lower priority). Tighter bounds can be achieved by arranging master and slaves to be neighbors in the network and/or providing

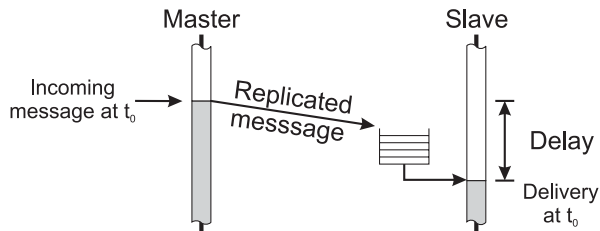


FIGURE 4: Master and slave separated by a fixed time delay but sharing a logical time reference.

dedicated links for master-slave coordination exchanges. Due to network variations, coordination messages can arrive ahead of time at the slave. Therefore, a *gated delivery queue* needs to hold the coordination messages until the designated delivery time. If network messages can arrive out-of-order, then the gated delivery queue must also support sorting of coordination messages by delivery time.

To complete the illusion of a single logical processor, only one core of the DMR pair should emit external outputs to the rest of the system. In the simplest manifestation, all outputs generated by a master are intercepted by its node controller and forwarded to its slave. A master’s output is compared against the corresponding output from the slave, and a single verified output is released to the rest of the system. This scheme is confluent with our requirement to verify the DMR outputs, as explained in the next section. However, this approach can interfere with normal interconnect flow control. In many flow control disciplines, a node is not allowed to receive a message unless it is assured that it can send a message in response to the received message. Because sending and receiving (on behalf of a logical node) are decoupled in LACROSS, the receiving master cannot directly sense network backpressure at the slave. The solution is a simple credit-debit system where the slave periodically informs the master of available space in its send buffers so the master knows, conservatively, when to stop receiving.

3.1 Distributed clock control

A globally synchronous clock in a distributed system is undesirable for many reasons and not strictly required by LACROSS. The only requirement in LACROSS is that the locally generated clocks at the master and slave do not drift to the point that the lag between the master and slave is insufficient to cover the worst-case transit latency of the coordination message. This condition can be detected when coordination messages begin to arrive at the slave too close to the delivery time minus some safety margin. In these cases, frequency control mechanisms such as down-spread spectrum modulation [21] are needed to actively slow down the slave clock to rebuild the master-to-slave delay. If necessary, large clock frequency adjustments (e.g., for thermal or power throttling) must be explicitly prepared with software assistance.

4 FAULT ISOLATION IN LACROSS

LACROSS provides redundant computation in the form of lock-stepped processors and caches; however, LACROSS must also locally detect and recover from soft errors and permanent faults. In this section, we discuss the integration of error detection into a cache coherence protocol, provide a protocol for recovery from various faults, and suggest a lightweight checkpointing solution for saving processor and cache state.

4.1 Soft error detection and recovery

We assume all on-chip architectural state (caches, architectural registers, condition codes, etc.) is protected from soft error. Recent work [35] proposed *fingerprinting* as a mechanism to detect transient errors in microarchitectural state and processor

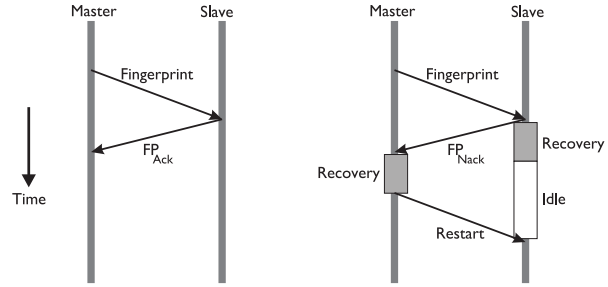


FIGURE 5: Soft-error detection (left) and recovery (right) protocols.

logic across a DMR pair. Each fingerprint consists of a hash of updates to architectural state. A single fingerprint captures the program execution up to the most recent committed instruction, and matching fingerprints across a DMR pair provides a high probabilistic guarantee that no soft error has occurred. Because each fingerprint is a small, 16-bit hash, frequent comparisons do not impose significant network bandwidth overheads.

To isolate errors to a single DMR pair, we require a fingerprint comparison before any change is made to external architectural state. In a simple implementation, any outgoing request or reply would come from the slave after comparing fingerprints to ensure the outgoing message was free from error. In Section 5, we examine performance optimizations that relax these simple requirements.

Because the slave operates behind the master in LACROSS, a natural point for error detection is at the slave, as illustrated in Figure 5(left). The master processor sends its slave a fingerprint and the slave responds with an acknowledgement indicating the master’s fingerprint matched its own. If the slave detects a mismatch in fingerprints, it initiates recovery.

Figure 5(right) illustrates the recovery protocol, where each DMR pair relies on a checkpoint of prior, correct state as the rollback-recovery point. Each fingerprint comparison verifies execution up to that point as correct; therefore, only a single checkpoint must be kept. Because the slave compares fingerprints for the pair, it also keeps the checkpoint for the pair. In the infrequent case of a fingerprint mismatch, the slave sends the checkpoint information to the master. The slave proceeds by restoring its state from the checkpoint and then waits for a signal from the master to resume execution.

Because every fingerprint comparison results in a new checkpoint and a fingerprint accompanies any irreversible modification to system state, only modifications to the processor state (registers, control codes, etc.) and on-chip caches need checkpointing. We discuss a low-overhead checkpoint implementation in Section 4.3.

4.2 Permanent fault detection and recovery

Permanent faults come in a variety of forms, ranging from single-transistor faults in the processor logic to the complete loss of a node. The system detects these faults by either observing repetitive fingerprint mismatches or detecting a timeout on fin-

gerprint or fingerprint acknowledgment. Because each node is built from a dual-core CMP, failure of an entire node affects two DMR pairs. Failure of one core in a node affects just one of the DMR pairs.

Two possible recovery actions exist. First, the functional member of each affected pair can continue to operate in a non-redundant mode until the failed node is replaced. This approach reduces coverage, exposing the system to further soft errors or permanent faults, but only requires additional hardware in the event a failure occurs. The second approach mitigates the loss of coverage by providing hot-spare DMR pairs that are automatically brought online to replace failed pairs. Whether the failed processor is a master or slave in the pair has no bearing on recovery from a permanent fault—the functional member of each pair copies its current state to the replacement pair.

4.3 Checkpoint creation

Recent interest in extending speculation support and enhancing reliability in microarchitecture has sparked a number of proposals for checkpoint mechanisms that recover beyond retirement [16, 23, 32, 37]. In general, our approach can work with any of these checkpoint mechanisms, provided the performance overhead of frequent checkpoint creation is negligible.

In this section, we suggest an example of a low-overhead recovery mechanism. In this approach, every checkpoint consists of a copy of the architectural registers and a log of previous cache values that were overwritten since the last checkpoint. Recent microarchitectures [17] maintain an architectural register file (ARF) of committed state. The ARF can be implemented as a redundant register file (RRF) structure to hold a checkpoint of architectural state by adding a shadow cell to each bit in the ARF SRAM [3, 14]. When a new checkpoint is created, the most recent value is flash-copied into the backup copy across the entire RRF structure, creating a checkpoint of the architectural registers in a single cycle.

Because each slave keeps a single checkpoint and creates new checkpoints frequently, a simple approach to cache checkpointing performs a copy-on-write in the cache on every store (as in previously-proposed mechanisms [16]). The previous value of the cache block is recorded in a FIFO checkpoint log, which, in the event of recovery, the slave replays in reverse to recreate the cache state. By executing ahead of the slave, the master keeps track of the FIFO log size and forces a fingerprint comparison before the log fills up. To remain in lockstep, the master and slave must both ‘create checkpoints’—that is, the master must perform the copy-on-write operations in the cache to reproduce any minor performance overheads the slave incurs.

5 PERFORMANCE OPTIMIZATIONS

The basic design for fault isolation in Section 4 has the master receive all incoming traffic and the slave send *all outgoing* requests or replies. In this way, all interprocessor communication experiences an extra latency corresponding to the master-slave delay. If this extra delay is manifested on the critical path

of computation, distributed redundant processing can result in a very significant performance overhead. In this section, we consider scenarios where a master processor can safely release outgoing requests or replies.

5.1 Relaxing read requests

Our first optimization takes advantage of the fact that not all outputs of the processor have an irreversible effect on system state. For example, a read request for a shared copy of a memory location does not alter the system state irreversibly. In these cases, the master still forwards the outbound request to the slave for corroboration but can at the same time release the request directly. If corroboration fails at the slave, the erroneous outputs either have no ill effect or can be reversed silently. For example, the shared read request to an erroneous address will cause the directory controller to respond with a shared copy of the cache block; however, the system loses no data as the requester’s cache and the directory retain the most up-to-date value.

On the other hand, outputs that alter the server’s system state must be verified as error-free before being released. A conceptually-simple case is non-idempotent loads or stores that modify peripheral devices (physical IOs). These requests cannot be released without corroboration, and the master-to-slave latency will be added to the critical path request latency. Accessing peripheral devices is slow—even relative to inter-processor communication—therefore, we expect little overhead from delaying these physical IOs.

A more subtle example consists of a request for a writable copy, which must be verified because such requests transfer sole ownership of a memory block to the requester. Improper handling leads to data loss. Fortunately, the additional latency in requesting a writable copy often does not impact the critical path because many mechanisms contribute to hide store latencies (e.g., relaxed memory models [16, 43]).

The scenario that is most likely to impact performance is when a node has to respond to a read request for a dirty cache line. This response’s output must be corroborated before being released by the slave. The extra delay in this request-and-response turnaround is reflected directly in the load latency of the requester and has a good chance of contributing to the overall critical path. This is often the case in commercial workloads, including online transaction processing (OLTP) and web servers, where migratory access patterns [5] result in frequent dirty reads, and low memory-level parallelism (MLP) [9] exacerbates the effect of the master-slave corroboration delay.

5.2 Validation filter

The impact of the master-to-slave delay on requests for dirty cache blocks can be avoided with the help of a *validation filter*, a mechanism similar in spirit to [25]. This hardware mechanism leverages the fact that, in most cases, the last store (production) comes long before the consuming read by another processor [36]. During this dead time, if the master and slave have compared even one fingerprint, then the dirty value in the master’s cache can be declared error-free. Therefore, when the

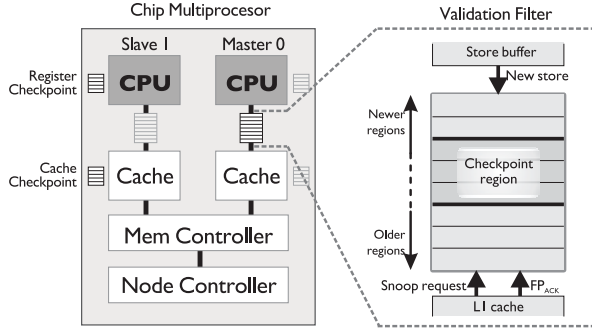


FIGURE 6: Proposed CMP additions: validation filter and checkpoints.

consuming read requests the dirty data, the master should respond directly without waiting for the slave. However, a more recently-written dirty cache location may still be pending fingerprint acknowledgement when a request for that block arrives. In these cases, the master must forward the requested dirty cache block value to the slave for corroboration and release. The validation filter is the mechanism at the master to help determine if a read request to a dirty cache block can bypass slave corroboration.

The validation filter tracks cache blocks that have been written but still await fingerprint acknowledgement. As shown in Figure 6, the filter is divided into regions corresponding to checkpoints pending slave acknowledgement. The validation filter allows fully associative lookup for the presence of a block by checkpoint regions.

The validation filter handles three events: (1) a new store from the processor, (2) a snoop on behalf of a remote dirty read request, and (3) a fingerprint acknowledgement from the slave.

1. When a store is written to the master cache, its address is also added to the most recent checkpoint region if its cache block is not already present in that region.
2. When a snoop request arrives, the validation filter checks for the presence of the cache block in any outstanding checkpoint region. If present, the response to the dirty read request must be forwarded to the slave for corroboration before release.
3. A fingerprint acknowledgement returned by the slave causes the validation filter to clear the blocks in the oldest checkpoint region.

Because of the need for fully-associative lookup, the validation filter capacity is limited (e.g., 64 entries), depending on available timing budget. In the event the validation filter overflows, the master can no longer properly distinguish between dirty cache blocks that are known error-free versus those pending fingerprint acknowledgement. After an overflow, the master must assume any request to a dirty cache block is to an unvalidated block and forwards the response on to the slave. To resynchronize after overflow, the master first clears the filter and then waits for the next checkpoint interval to start before logging new stores again. The validation filter resumes normal snoop handling only after all fingerprints sent prior to the overflow have been acknowledged.

6 EVALUATION

We evaluate the error-free performance of LACROSS with FLEXUS, a cycle-accurate full-system simulator [20]. We model a 16-node DSM running Solaris 8, where each dual-core node contains two speculative, 8-way out-of-order superscalar processor cores and an on-chip private cache hierarchy.

Our performance evaluations compare with a DSM of tightly lockstepped DMR pairs where each CMP operates as one redundant pair as illustrated previously in Figure 3. This baseline has the same infrastructure cost as our distributed DMR approach, but removes the overhead due to distributing the lockstepped pairs. We use this baseline solely for performance comparison—the tightly-coupled pairs cannot tolerate node loss.

Directory state at each node is maintained with a microcoded controller, and the nodes communicate through a NACK-free, 3-hop cache-coherence protocol derived from Piranha [4]. We model an interconnect based on the HP GS1280 [26]. We implement the total store order (TSO) memory consistency model [43] and perform speculative load and store prefetching [15]. Other relevant system parameters are listed in Table 1.

We evaluate four commercial workloads and three scientific applications. We run both IBM DB2 v8 ESE and Oracle 10g Enterprise Database Server on an online transaction processing (OLTP) workload modeled after a 100 warehouse TPC-C installation. We evaluate web server performance using the SpecWeb99 workload running on both Apache HTTP Server v2.0 and Zeus Web Server v4.3. In addition to these server benchmarks, we evaluate three scientific workloads that show a range of computation and memory access patterns: em3d is a read-intensive benchmark dominated by memory access latency; ocean contains short bursts of parallel memory accesses followed by long computation phases; and gauss models a small, in-cache kernel dominated by coherent read miss latency.

For all workloads except gauss, we use a systematic sampling approach derived from SMARTS [45]. We collect approxi-

TABLE 1. DSM Server configuration.

Processing Cores	UltraSPARC III ISA 4 GHz 8-stage pipeline; out-of-order 8-wide dispatch / retirement 256-entry ROB; 64-entry store buffer
L1 Caches (per core)	Split I/D, 64KB 2-way, 2-cycle load-to-use 4 ports, 32 MSHRs
L2 Caches (per core)	Unified, 8MB 8-way, 25-cycle hit latency 1 port, 32 MSHRs
Main Memory	60 ns access latency 64 banks per node 64-byte coherence unit
Protocol Controller	1 GHz microcoded controller 64 transaction contexts
Interconnect	4x4 2D torus 25 ns latency per hop 128 GB/s peak bisection bandwidth

TABLE 2. Workload configuration.

<i>Commercial Applications</i>	
OLTP/DB2	100 warehouses (10 GB), 64 clients, 450 MB BP
OLTP/Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web/Apache	16K connections, fastCGI
Web/Zeus	16K connections, fastCGI
<i>Scientific Applications</i>	
em3d	3M nodes, degree 2, 15% remote, span 5
ocean	1026x1026 grid, 600s rel., 20km res., 1e-07 err
gauss	256-by-256 matrix reduction

mately 100 brief measurements of 50,000 cycles each. We launch all measurements from checkpoints with warmed caches, branch predictors, and directory state, then run for 100,000 cycles to warm queue and interconnect state prior to collecting statistics. We include 95% confidence intervals. We aggregate total cycles per user instruction as CPI, which is inversely proportional to overall system throughput [44]. In gauss, we measure the user-mode CPI of the slowest processor over an iteration, which reflects the overall execution time for the workload.

6.1 DMR Coordination

We isolate performance impact due to the additional load of coordination messages by allowing the master to respond to all coherence messages directly without corroborating with the slave. LACROSS introduces additional traffic in the network link between master and slave, effectively doubling the number of packets sent in the network. Every time the master receives any external input from the system, a coordination message is sent to the slave.

Figure 7 shows the performance overhead from LACROSS coordination, reported as cycles per user instruction normalized to the baseline. On average, a 4.4% overhead in performance occurs solely due to the additional network contention. Most of the workloads we studied have relatively low network utilization in the baseline system.

Oracle/OLTP, the workload most sensitive to the additional LACROSS traffic, offers 0.22 packets/cycle of network load in the baseline system, which is slightly less than half the 0.55 packets/cycle of available network throughput, as measured with uniform random packet injection [13]. LACROSS doubles the offered load, thereby approaching the available throughput and increasing latency on the lowest priority virtual channels

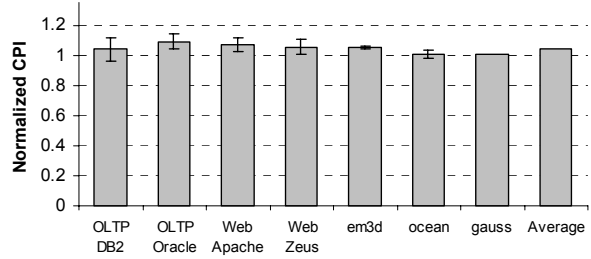


FIGURE 7: Overhead from coordinating DMR pairs.

where normal coherence requests travel. In OLTP/Oracle, we observe a 38% average increase in round-trip network latency, which corresponds to an 18% increase in time spent on remote misses and the 9% CPI overhead.

6.2 Fault Isolation

To show the performance effects of master-slave corroboration delay without contributions from increased coordination traffic, we take the converse approach from Section 6.1 by providing dedicated network links for all master-to-slave connections.

Relaxing Read Requests. We first consider the case where requests for dirty data must wait for the slave to compare fingerprints, which imposes an additional master-to-slave delay on the latency of the request. The top three lines in Figure 8 show the overhead from fault isolation as a function of the master-to-slave delay. The observed increases are attributable to the increase in dirty coherence miss latency and the commensurate queuing effects in the system.

The minimal master-to-slave delay for our network model is 550 processor clock cycles, which is a function of the buffering in the network switches. Increasing the buffering increases the network’s ability to tolerate bursts of data transfers, but also increases the worst-case latency.

A master-to-slave delay of 550 cycles increases the average dirty miss latency by nearly 50% in our system. On average, the four commercial workloads spend 19% of total time waiting for dirty coherence misses. Although em3d and ocean spend a negligible fraction of time (less than 3%) on dirty misses, gauss spends 16% of total time on dirty misses due to frequent barrier synchronization and pivot computation. Clearly, scaling the master-to-slave delay has a major impact on system performance.

Validation Filter. The bottom three lines in Figure 8 show that the validation filter effectively removes the sensitivity to the master-to-slave delay. What little performance overhead

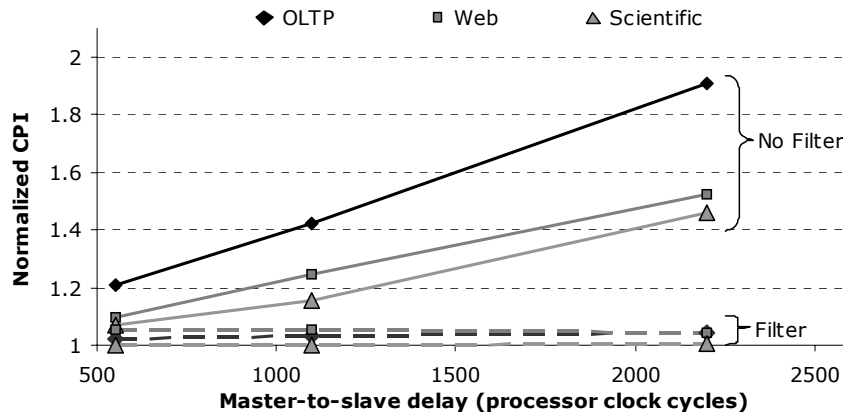


FIGURE 8: Performance overhead as a function of the master-to-slave delay.

remains is attributable to lock transfers or barrier synchronization—cases where cache blocks are consumed immediately after production. Across all the applications studied, over 98% of all dirty misses hit in the validation filter—that is, the master can reply to 98% of all requests for dirty data without waiting for the slave to compare fingerprints. Limiting the filter to 64 entries does not impact the hit rate—the filter overflows less than 0.02% of the time in the worst-case (OLTP/Oracle).

6.3 Delaying physical IOs

In this section, we highlight the impact of delaying only physical IOs in the system. We remove all other sources of LACROSS-based performance overhead and scale the latency of memory-mapped accesses to pages that the MMU indicates have side effects. Table 3 shows the normalized slowdown as a function of increasing master-to-slave delay. We do not report results for the scientific applications, which perform no device IO.

We found that little sensitivity exists to delaying physical IOs, even in IO-intensive commercial workloads. In the worst case, OLTP suffers just 6.8% slowdown. Note that we do not stall the delivery of interrupts to a processor or delay DMA activity—only the reading or writing of peripheral device registers. In practice, a disk or network driver spends little time accessing a controller’s memory-mapped registers. The majority of time goes to setting up transfers or processing data from a DMA buffer, which are unaffected in our design.

TABLE 3. Slowdown when delaying physical IOs.

	Master-to-slave delay (processor clock cycles)		
	550	1100	2200
OLTP	1.1%	2.2%	6.8%
Web	0.2%	1.2%	3.8%

6.4 Full system performance

Figure 9 shows the overall LACROSS performance by combining distributed DMR pairs with the fault isolation protocols. We show the performance overhead for the minimal choice of master-to-slave delay (550 processor cycles). Because the validation filter optimization requires non-trivial changes to the processor core interface, we include overall results with and without the filter.

Without the validation filter, OLTP/DB2 incurs the highest performance overhead of 21%, due primarily to the frequent, migratory sharing of data, which incur additional latency from the master-to-slave delay. With the validation filter, the performance overhead in OLTP/DB2 is reduced to 6.5%, and the worst-case workload becomes OLTP/Oracle, which incurs a 13% performance overhead. Most of this overhead is due to network contention overhead from coordination messages. The remaining contributions come from delaying physical IOs and imposing the master-to-slave delay on highly-contended blocks.

On average, the system performs within 13% of the baseline without the validation filter and within 5.8% of the baseline with the filter. Although the improvements from the filter seem small, the real benefit comes from decoupling the performance overheads from the master-to-slave delay, as shown previously in Figure 8.

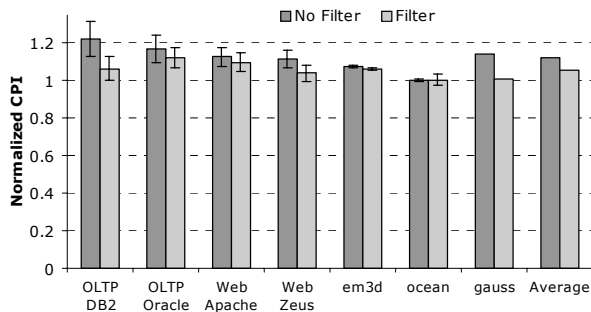


FIGURE 9: Full-system overhead in LACROSS.

7 RELATED WORK

At least four decades of development have been devoted toward improving reliability and availability for mainframe systems. IBM's z-series [34] machines include processor designs that detect and recover from virtually any transient error, and the system transparently provides redundant hardware for many permanent faults. Our contribution is to enable distributed redundancy that survives chip failure, a critical reliability factor as device scaling continues.

The HP (formerly Tandem) NonStop systems [6] traditionally constructed fail-fast DMR pairs from tightly lockstepped processors. When an error is detected, the pair is assumed dead and the process is moved to a spare DMR pair to resume execution. This so-called 'pair-and-spare' approach requires software assistance to provide the spare process with sufficient state information to intervene should an error be detected. Our design obviates the need for specialized software—the hardware provides software-transparent error recovery.

Our approach is inspired by the observation that cache coherence protocols for DSM multiprocessors have to deal with a multitude of race conditions and failure scenarios. Sun Microsystems' S3.mp system [29] included extensions to the coherence protocols to provide fault isolation and memory replication. Stanford's Dash DSM multiprocessor also was used to explore fault confinement capabilities [41] that contributed insights to our approach. We extended these ideas to add dual-modular redundancy for transparent recovery.

Numerous proposals exist for creating coarse-grained, global checkpoints of DSM architectural state for the purposes of surviving processor or node failures [1,24,32]. Our work advances these proposals by enabling local error detection and recovery at fine-grained intervals as required by IO-intensive workloads [35].

The idea of using a time-shifted slave processor for redundant checking has been employed before in Slipstream processors [39], simultaneous redundant threading [33,42], chip-level redundant threading [19,28], and microarchitecture checkers [2]. However, these proposals rely on close physical proximity for the master and slave cores, so that latency and bandwidth issues are fairly benign. Our contribution is the development of a viable system that allows the physical distribution of the DMR pairs across a system area network.

8 CONCLUSIONS

This paper presented a DSM system design that supports lockstep operation of physically distributed processing modules. The proposed approach to distributed redundancy provides reliability in the presence of transient or permanent faults in the processing modules without sacrificing the scalability of the underlying DSM architecture. This paper addressed two key design challenges: the coordination of a master and slave DMR pair and the corroboration of the decoupled redundant processing modules. Evaluations based on a range of commercial and scientific workloads showed that in the worst case, the error-free performance overhead of distributed lockstepping is within 13% (5.8% on average) of a baseline system.

ACKNOWLEDGEMENTS

The authors would like to thank the members of the TRUSS research group at CMU for their feedback on earlier drafts of this paper and the CMU SimFlex team for simulation infrastructure. This work was funded in part by NSF awards ACI-0325802 and CCF-0347560, Intel Corp., the Center for Circuit and System Solutions (C2S2), the Carnegie Mellon CyLab, and fellowships from the Department of Defense and the Alfred P. Sloan Foundation.

References

- [1] R. E. Ahmed et al. Cache-aided rollback error recovery CARER algorithm for shared-memory multiprocessor systems. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, June 1990, 82–88.
- [2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, November 1999.
- [3] D. Balkan, et. al. Selective writeback: Improving processor performance and energy-efficiency. In *1st IBM Watson conference on Interaction between Architecture Circuits and Compilers*, October 2004.
- [4] L. Barroso et al. Piranha: A scalable architecture base on single-chip multiprocessing. In *Proc. of 27th Intl. Symp. on Comp. Arch. (ISCA-27)*, June 2000.
- [5] L. A. Barroso et al. Memory system characterization of commercial workloads. In *Proc. of 25th Intl. Symp. on Comp. Arch. (ISCA-25)*, June 1998, 3–14.
- [6] W. Bartlett and B. Ball. Tandem's approach to fault tolerance. *Tandem Systems Rev.*, 8:84–95, February 1988.
- [7] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *Keynote address, MICRO-37*, 2004.
- [8] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–17, November-December 2005.
- [9] Y. Chou et al. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of 31st Intl. Symp. on Comp. Arch. (ISCA-31)*, June 2004.
- [10] I. Corporation. Intel E8500 chipset north bridge (nb), March 2005.
- [11] Z. Cvetanovic. Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor. In *Proc. of 30th Intl. Symp. on Comp. Arch. (ISCA-30)*, June 2003, 218–229.
- [12] T. J. Dell. A white paper on the benefits of chipkillcorrect ecc for pc server main memory. In *IBM Whitepaper*, 1997.
- [13] J. Duato et al. *Interconnection Networks: an Engineering Approach*. Morgan Kaufmann, 2003.
- [14] O. Ergin, et. al. Increasing processor performance through early register release. In *Proceedings of ICCD*, 2004.
- [15] K. Gharachorloo et al. Two techniques to enhance the performance of memory consistency models. In *Proc. of the 1991 Intl. Conf. on Parallel Processing*, Aug. 1991, 1–35–364.
- [16] C. Gniady et al. Is SC + ILP = RC? In *Proc. of 26th Intl.*

- Symp. on Comp. Arch. (ISCA-26)*, May 1999, 162–171.
- [17] S. Gochman et al. The intel pentium m processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.
- [18] B. T. Gold et al. TRUSS: A reliable, scalable server architecture. *IEEE Micro*, 25(6), Nov-Dec 2005.
- [19] M. Gomaal et al. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.
- [20] N. Hardavellas et al. SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Performance Evaluation Review*, 31(4):31–35, April 2004.
- [21] K. Hardin et al. Design considerations of phase-locked loop systems for spread spectrum clock generation compatibility. In *Proc. Intl. Symp. on Electromagnetic Compatibility*, 1997.
- [22] J. Hennessy. Back to the future: Time to return to long-standing problems in computer systems? In *Keynote address, FCRC*, 1999.
- [23] J. F. Martinez et al. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proc. of 35th IEEE/ACM Intl. Symp. on Microarch. (MICRO 35)*, Nov 2002, 3–14.
- [24] C. Morin et al. An efficient and scalable approach for implementing fault-tolerant dsm architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.
- [25] A. Moshovos et al. JETTY: filtering snoops for reduced energy consumption in SMP servers. In *Proc. of Seventh IEEE Symp. on High-Perf. Comp. Arch.*, Mar. 2001.
- [26] S. S. Mukherjee et al. The alpha 21364 network architecture. *IEEE Micro*, 22(1), Jan-Feb 2002.
- [27] S. S. Mukherjee et al. The soft error problem: an architectural perspective. In *Proc. Intl. Symp. on High-Performance Computer Architecture*, 2005.
- [28] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002, 99–110.
- [29] A. G. Nowatzky et al. The s3.mp scalable shared memory multiprocessor. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, Jan 1994, 144–153.
- [30] D. Patterson. Recovery oriented computing: A new research agenda for a new century. In *Keynote address, HPCA-8*, 2002.
- [31] D. Patterson et al. A case for redundant arrays of inexpensive disks (raid). In *Proc. of 7th ACM Intl. Conf. on Management of Data (SIGMOD '88)*, June 1988, 109–116.
- [32] M. Prvulovic et al. ReVive: cost-effective architectural support for rollback recovery in shared memory multiprocessors. In *Proc. of 29th Intl. Symp. on Comp. Arch. (ISCA-29)*, June 2002.
- [33] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.
- [34] T. J. Slegel, et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March/April 1999.
- [35] J. C. Smolens et al. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proc. of Eleventh Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS XI)*, Boston, Massachusetts, Oct. 2004. 224–234.
- [36] S. Somogyi et al. Memory coherence activity prediction in commercial workloads. In *Proc. of Third Workshop on Memory Perf. Issues (WMPI-2004)*, June 2004.
- [37] D. J. Sorin et al. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of 29th Intl. Symp. on Comp. Arch. (ISCA-29)*, June 2002.
- [38] J. Srinivasan et al. The impact of technology scaling on lifetime reliability. In *Proc. Intl. Conf. on Dependable Systems and Networks*, 2004.
- [39] K. Sundaramoorthy et al. Slipstream processors: improving both performance and fault tolerance. In *Proc. of Ninth Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS IX)*, Nov. 2000.
- [40] P. Tadayon. Thermal challenges during microprocessor testing. *Intel Technology Journal*, 4(3), August 2000.
- [41] D. Teodosiu et al. Hardware fault containment in scalable shared-memory multiprocessors. In *ISCA97*, June 1997.
- [42] T. N. Vijaykumar et al. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.
- [43] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.
- [44] T. F. Wenisch et al. Temporal streaming of shared memory. In *Proc. of 32nd Intl. Symp. on Comp. Arch. (ISCA-32)*, June 2005.
- [45] R. Wunderlich et al. SMARTS: Accelerating microarchitecture simulation through rigorous statistical sampling. In *Proc. of 30th Intl. Symp. on Comp. Arch. (ISCA-30)*, June 2003.