

Chip-Level Redundancy in Distributed Shared-Memory Multiprocessors

Brian T. Gold^{1*}, Babak Falsafi², and James C. Hoe¹

¹*Computer Architecture Laboratory (CALCM), Carnegie Mellon University*

²*Parallel Systems Architecture Lab (PARSA), École Polytechnique Fédérale de Lausanne*
<http://www.ece.cmu.edu/~truss>

Abstract—Distributed shared-memory (DSM) multiprocessors provide a scalable hardware platform, but lack the necessary redundancy for mainframe-level reliability and availability. Chip-level redundancy in a DSM server faces a key challenge: the increased latency to check results among redundant components. To address performance overheads, we propose a *checking filter* that reduces the number of checking operations impeding the critical path of execution. Furthermore, we propose to decouple checking operations from the coherence protocol, which simplifies the implementation and permits reuse of existing coherence controller hardware. Our simulation results of commercial workloads indicate average performance overhead is within 4% (9% maximum) of tightly coupled DMR solutions.

I. INTRODUCTION

Mainframes remain key to business-critical operations that range from transaction processing and decision support to supply-chain management and logistical operations. Unfortunately, increasing levels of integration result in rising soft- and hard-error rates for future processors [4,12]. For mainframes in particular, the challenge becomes balancing performance enhancements with design complexity while maintaining high reliability, availability, and serviceability (RAS) targets.

To mitigate increasing error rates, recent work advocates tightly coupled checkers in the form of redundant threads [15,24], cores [8,13], or dedicated checking logic [1,20]. Although these designs impose little performance overhead, tightly coupled redundancy does not protect from complete chip failure, which research predicts will occur more frequently with continued device scaling [23].

Replicating execution across dual-modular redundant (DMR) chips provides protection from chip-level failures [2]. Furthermore, redundant execution at the chip level reduces or eliminates the need for complex microarchitectural changes required by tightly coupled checkers. However, existing chip-level redundant architectures all have significant disadvantages: either they require custom, message-passing software [2] or compromise scalability by using a small, broadcast-based interconnect [17].

In contrast, hardware distributed shared-memory (DSM) makes an ideal platform for building future mainframe servers. DSM preserves the familiar shared-memory programming model and provides a scalable hardware architecture. Integrating physically distributed DMR chips in a hardware DSM, however, faces a fundamental performance challenge due to the long latency—hundreds or thousands of processor clock cycles—required to check execution results.

Our prior proposal [7] for a chip-level redundant DSM suffers from unacceptable performance overhead, particularly in commercial workloads, due to frequent, long-latency checking on the critical path of execution. Previously proposed mechanisms [7] that reduce the latency of checking fail to provide adequate improvement and require impractical changes to the cache coherence protocol and its implementation.

To overcome these limitations from prior work, we propose the following key mechanisms for chip-level redundancy in a DSM server:

- **Checking filter.** The majority of modified cache blocks remain on chip long after the last update to the block [11]. Because of this “dead time”, we observe that over 90% of checking operations can be removed from the critical path of execution. We propose a hardware mechanism called the *checking filter* to determine when unnecessary checks can be elided from the critical path. We show that a checking filter can reduce performance overheads to within 4% on average (9% worst case), when compared with a tightly coupled DMR system.

- **Decoupled checking protocol.** We propose to decouple error checking from the DSM coherence protocol. Unlike our previous design [7], decoupled checking requires no modification to the existing coherence controller. Although the checking latency increases slightly with a decoupled design, the effectiveness of the checking filter is sufficiently high that overall performance overhead not affected.

We implement the checking filter and decoupled checking protocol in a scalable DSM architecture, and we provide a detailed description of the hardware design, including lockstep coordination, error checking, and rollback-recovery operations. Our evaluations, using cycle-accurate simulation of commercial workloads, show performance overhead is within 9% of tightly coupled redundancy.

* Now with Sun Microsystems Labs, Menlo Park, CA

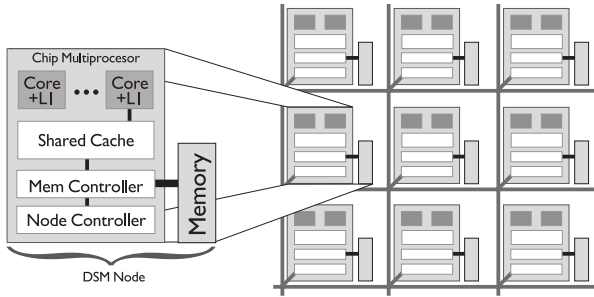


Figure 1. DSM system model.

Paper Outline. In Section II we review prior work on distributed DMR in hardware DSM servers. Section III proposes the checking filter, and Section IV describes the decoupled checking protocol. In Section V, we present details of our system design, which we evaluate in Section VI. We present related work in Section VI and conclude in Section VII.

II. BACKGROUND

DSM makes an ideal platform for future mainframes where scalable processing and memory capacity are required within a shared-memory programming model. Moreover, in comparison with conventional, monolithic memory systems (e.g., SMP), modern DSM architectures reduce the latency to memory by integrating DRAM controllers and network interface hardware directly on chip. Figure 1 illustrates a DSM architecture built from chip multiprocessors (CMPs) with integrated memory and interconnect controllers.

We apply a single-point-of-failure model for transient and permanent faults affecting the CMPs of our system, where a single fault (e.g., radiation strike [12] or device wearout [4,23]) may alter any number of state or logic elements of the chip, including complete chip failure. We assume that the processor datapath is currently unprotected, but require that information redundancy (e.g., parity or ECC codes) protect architectural state such as register files and caches. Furthermore, integrated coherence and memory controllers are protected by self-checking circuits [17] and/or end-to-end protocol checks [21]. Orthogonal techniques protect components outside the CMPs, such as DRAM [5], interconnect [22], and I/O devices [14].

To mitigate faults in processor cores and caches, traditional lockstep designs [2] place pairs of redundant chips in close physical proximity and use a common clock to drive each pair. Each pair shares a single bus such that external inputs (e.g., interrupts, cache line fills, etc.) arrive simultaneously in both chips. A dedicated checker circuit compares outgoing data before the data gets released to the outside system.

In a distributed DMR design, however, lockstep pairs cannot be driven with a common clock or share a single input bus. Previous work [7] addresses these challenges by time-shifting DMR pairs, where each ‘master’ chip operates a fixed time reference ahead of its redundant ‘slave’. To avoid coordinating recovery with other DMR pairs or the outside system, each DMR pair creates a lightweight checkpoint of locally modified architectural state any time the pair interacts with the outside system.

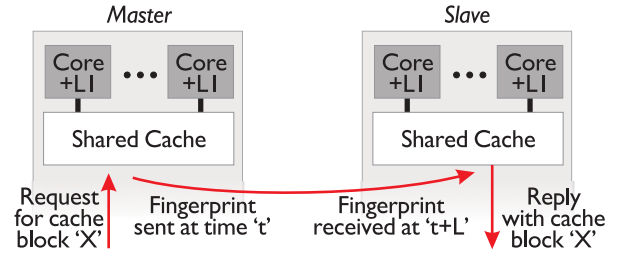


Figure 2. Checking in TRUSS [7], extended to CMPs.

After receiving an external input, before creating a checkpoint, a DMR pair first compares signatures of computation results, called *fingerprints* [19], which efficiently detect errors that affect retired instructions. Figure 2 illustrates the checking operations required if the input is a request for modified architectural state, termed a *dirty read request* in shared-memory systems. The master cannot reply to the request directly, because the master cannot guarantee the requested cache block (‘X’) is free. Instead, the reply must wait for the master/slave time shift (‘L’) and the subsequent fingerprint comparison at the slave.

Previous work [7] suffered from unacceptable performance overheads due to the long-latency checking imposed on dirty read operations. If the master can differentiate previously checked cache blocks from unchecked cache blocks, the master can reply directly to the majority of dirty read requests. In Section III, we propose the checking filter as a mechanism to specifically detect accesses to unchecked blocks.

Figure 2 also shows the second key shortcoming of our previous design: integrated cache coherence and checking protocols. The master and slave coherence controllers are not redundant as they fundamentally implement different behaviors. For the slave to reply to request ‘X’, it must implement a new coherence protocol and bypass the existing protocol implementation.

Adding a new protocol controller for redundant execution is impractical. Existing coherence protocol implementations are complex, hard-to-verify structures [6]. A practical implementation of distributed DMR must therefore decouple checking from coherence, as we propose in Section IV.

III. CHECKING FILTERS

Most dirty cache blocks are updated long before an external coherence request accesses the modified value [11]. Leveraging the fact that fingerprints summarize *all* previously retired instructions [19], we observe that most cached values are effectively checked by prior fingerprint comparisons. We propose a checking filter, kept at each master processor core, which identifies unchecked cache blocks requested by dirty read requests and permits the master to reply without first checking with the slave.

A. Filter Operation

Conceptually, the checking filter operates as a searchable queue, where each entry consists of addresses from globally visible stores occurring between two fingerprints (a *fingerprint*

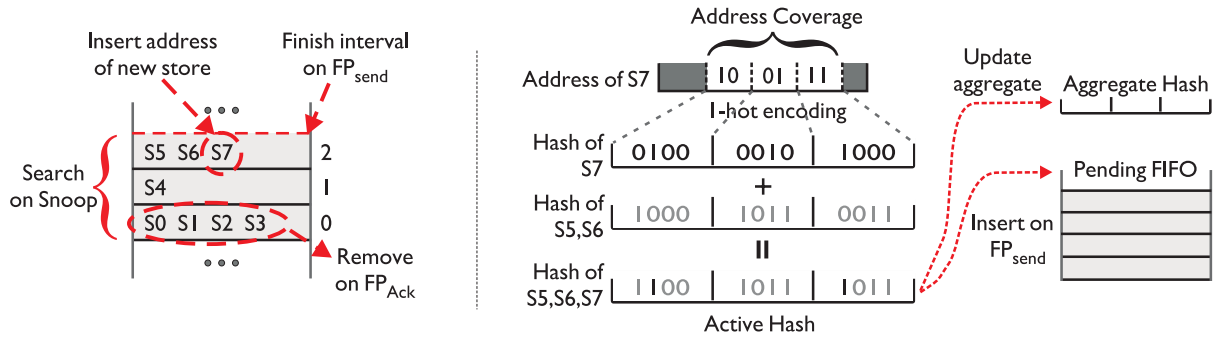


Figure 3. (left) Conceptual operation of checking filters, (right) hardware implementation with Bloom Filters

interval). Figure 3(left) illustrates the abstract operation of the checking filter. Stores are marked with sequence numbers (e.g., ‘S0’ is older than ‘S1’), and each row in the queue is marked on the right with a fingerprint interval number.

The filter must support four operations: (1) a new store, (2) a new fingerprint being sent, (3) a fingerprint acknowledgement from the slave, and (4) a snoop on behalf of a dirty read request. We examine each of these as follows:

1. When a store instruction becomes globally visible, its cache block address is added to the most recent checking interval.

2. When a new fingerprint is sent (FP_{send}), the filter closes the previous checking interval and creates a new interval in which stores (operation 1) are added. The addresses being kept previously in the “new interval” are now part of the pending intervals.

3. When the slave acknowledges a fingerprint (FP_{ack}), the filter removes the addresses in the oldest checking interval.

4. When a snoop request arrives, the filter checks for the presence of the requested cache block anywhere in the queue. If present, the response to the dirty read request must wait for the slave to corroborate its contents before release. If absent, the master can reply to the request directly, thereby eliding the long-latency check.

B. Filter Design

Implementing the conceptual organization shown in Figure 3(left), the checking filter requires a CAM structure with age-based lookup, similar in complexity to a load-store queue (LSQ) in an out-of-order processor core. We observe that the filter only needs to indicate when a block *must* be checked. False positives—where the filter incorrectly enforces checking on a previously checked block—only suffer performance penalty of checking.

We propose a design based on Bloom Filters [3], which indicate presence of an item (a cache block address) by mapping its key (a portion of the address) to a sequence of bits. To avoid name conflicts with our checking filter, we refer to the set of bits in a Bloom Filter structure as a “hash”.

Figure 3(right) illustrates the design we propose. The block address is subdivided into a number of segments, shown in white (the gray portions are unused). Each segment is encoded using a one-hot encoding to create the hash output. We keep an *active hash* for store addresses in the current fingerprint interval. As they become visible, new stores are added

to the active hash. When a new fingerprint is sent to the slave, the active hash is inserted in a pending hash FIFO queue. For efficient lookup, we keep an additional *aggregate hash* that consists of a logical OR of all pending hashes, including the active hash. When a fingerprint acknowledgement arrives from the slave, the oldest hash from the pending queue is removed and the aggregate hash is rebuilt by logically ORing the pending hashes and active hash.

The systems we study use the Total Store Order (TSO) memory model, which enforces program order among stores from retirement until global visibility in the L1 cache. As in commercial TSO designs, we use a store buffer to preserve this order. Because the checking filter also preserves the order among stores across fingerprint intervals, we augment the store buffer to include a fingerprint interval counter with each entry. We maintain a separate counter of fingerprint acknowledgements from the slave. When a store drains from the store buffer, its address is placed in the active hash if the store’s fingerprint counter is greater than the last acknowledged-fingerprint counter. The counter size is bounded by twice the maximum number of in-flight fingerprints. In practice, 64 or fewer fingerprints ever exist at once, so 7-bit counters are sufficient.

IV. CHECKING PROTOCOL

A. Coupled checking

To reduce the penalty applied to coherence requests, our prior proposal coupled the coherence protocol and checking operations. Figure 4(left) illustrates the coupled nature of this approach. When a dirty read request arrives at the master and accesses an unchecked cache block, the master forwards the request and the current fingerprint to the slave. The slave, after reaching the same point in execution, replies to the request if the slave’s fingerprint matches the master’s.

By sending the reply directly from the slave, coupled checking minimizes the penalty incurred by coherence requests when a check is required. The implementation of the coherence protocol, however, is prohibitively complex. The master and slave coherence controllers do not operate redundantly; therefore, the existing coherence controller on the slave cannot be used. The combined protocol must be verified [6] and a new coherence controller must be implemented at the slave. The slave’s controller must accept state updates from the master’s

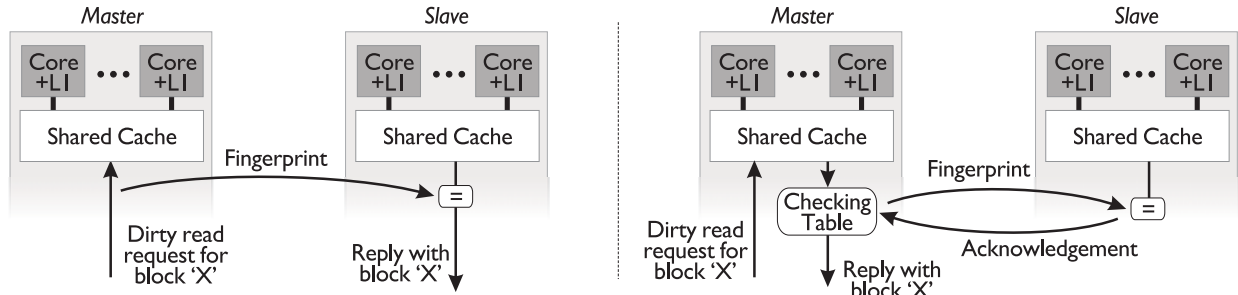


Figure 4. (left) Coupled checking, (right) decoupled checking.

coherence controller, which inform the slave where to reply to coherence requests.

B. Decoupled checking

In contrast, decoupled checking (Figure 4(right)) separates the coherence protocol implementation from checking operations. After comparing fingerprint values, the slave responds to the master with an acknowledgement, and the master sends any appropriate coherence replies. The acknowledgement step introduces an additional network hop to the checking penalty. Although this acknowledgement exists in the coupled protocol as well, it is not on the critical path of the reply.

The decoupled checking protocol permits the use of a coherence protocol controller from a non-redundant design without modification. To implement the decoupled protocol, we introduce a new hardware structure, the *checking table*, between the cache and existing coherence controller. The checking table can be simply bypassed if the chip is not used in a master/slave configuration (e.g., non-redundant).

The checking table is responsible for holding memory operations that require an explicit check with the slave. When a reply requiring a check is sent from the cache to the coherence controller (e.g., containing the dirty cache block being requested), the checking table will force a new fingerprint to be sent and insert the reply into a list of stalled operations. The stalled reply is released when an acknowledgement, tagged with the same address, is received from the slave.

V. LOCKSTEP COORDINATION

When the DMR modules are situated on the same specially-designed chip or motherboard, they 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 enforcing lockstep is infeasible in a distributed DMR arrangement. Instead, our solution takes an asymmetrical approach to lockstepping where true simultaneity is not needed.

Instead, we opt for a time-shifted lockstep, where the precise microarchitectural behaviors are duplicated on master and slave chips, but with a constant time delay separating them. There are three key aspects to maintaining this time shift: timestamps, input replication, and deterministic behaviors.

Timestamp counters maintain a time reference for both master and slave—the time-shift between master and slave is measured in timestamp counter ticks. They need not be driven by tightly synchronized physical clocks. Rather, they mark the

progress of execution on the cores of the CMP. The implementation of a timestamp counter is dependent on the specific details of the chip design and is beyond the scope of this paper.

Input replication ensures that both master and slave observe identical input stimuli. In our design, input replication is actively maintained with *coordination messages*, which are illustrated in Figure 5. When the master receives an external input, it records the current timestamp counter value and sends both a copy of the input data and the timestamp value to the slave. The slave, running a fixed timestamp delay behind the master, places the coordination message in a FIFO called the *gated delivery queue*. The gated delivery queue sends queued inputs to the slave when its local timestamp matches the message’s timestamp plus the fixed delay.

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). Messages that arrive ahead of the worst-case transit latency are buffered in the gated delivery queue, whose size is bounded by the maximum buffering in one network hop.

Because we replicate the payload and timing of external inputs, we require deterministic behavior for the processor cores and caches. Fortunately, deterministic behavior is also a requirement for many testing approaches, including the signature-mode scan found in Intel processors [10]. Furthermore, recent work [16] shows that deterministic behaviors is possible on CMPs with source-synchronous messaging across multiple clock domains.

Note that a globally synchronous clock in a distributed system is undesirable for many reasons and not required here. The only requirement is that the locally generated clocks at the master and slave do not drift to the point that the lag between

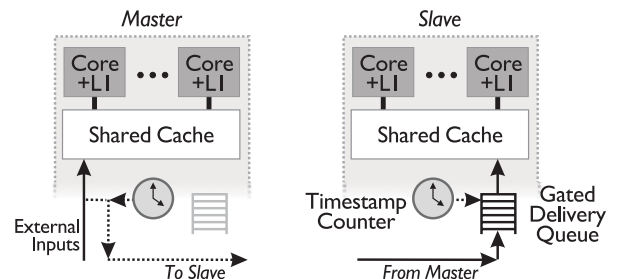


Figure 5. Lockstep coordination hardware.

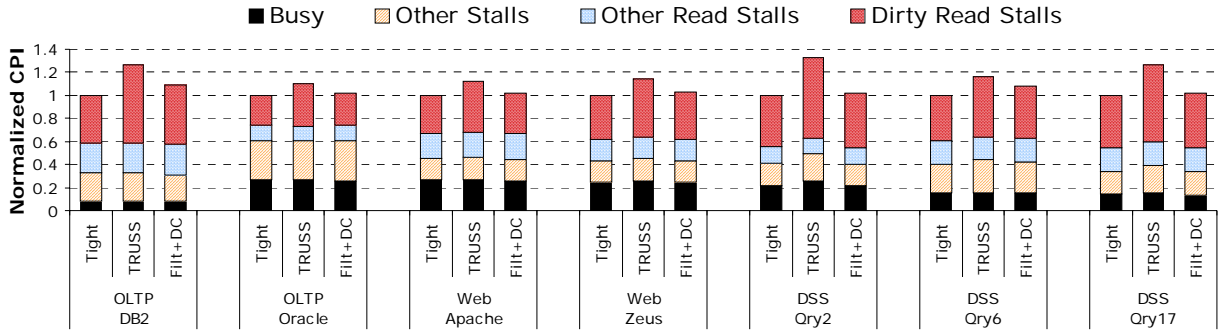


Figure 6. Baseline time breakdowns.

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 [9] are needed to slow down the slave clock and 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.

VI. EVALUATION

We evaluate error-free performance with FLEXUS, a cycle-accurate full-system simulator [25]. We model a DSM with 16 logical processors (16 nodes for tightly coupled DMR and 32 nodes for distributed DMR). Each node contains a speculative, 4-way out-of-order superscalar processor core and an on-chip cache hierarchy. We chose a single-core baseline to facilitate comparisons with previous work [7] and because the results shown here are sensitive to scaling the number of nodes, not cores. Other relevant system parameters are listed in Table 1.

Our performance evaluations compare with a DSM of tightly lockstepped DMR pairs, where each chip contains a lockstepped checker circuit (as in [18]). We assume the lockstepped checker introduces no performance overhead; however, we use this baseline solely for performance

comparison—the tightly-coupled pairs cannot tolerate node loss. Our design doubles the number of nodes over a non-redundant or tightly lockstepped DMR system.

Table 2 lists parameters of the workloads used in our evaluations. For all workloads, we use a systematic sampling approach [25] that draws approximately 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 aggregate total cycles per user instruction as CPI, which is inversely proportional to overall system throughput [25].

A. Baseline Performance

We evaluate the baseline performance of our workload suite with a tightly coupled DMR system, labeled “Tight” in Figure 6. The large fraction of time spent on dirty reads indicates that previous designs (“TRUSS”) will have significant additional stalls due to checking overheads. As expected, Figure 6 shows that the peak overhead is 33% (DSS Query 2) and the average overhead is 19%.

Figure 6 also shows the optimal effectiveness of the checking filter and the impact from decoupled checking, labeled “Filt+DC”. We simulated an oracle filter implementation that produces no false positives—all checking stalls are due to true data races in the programs. The decoupled checking imposes an additional network hop on all checked operations. In our model, the additional network hop results in a 7% increase in latency on dirty reads.

TABLE 1. DSM SERVER CONFIGURATION.

Processing Cores	UltraSPARC III ISA 4 GHz 8-stage pipeline; out-of-order 4-wide dispatch / retirement 256-entry ROB; 64-entry store buffer
L1 Caches	Split I/D, 64KB 2-way, 2-cycle load-to-use 4 ports, 32 MSHRs
L2 Cache	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 (baseline), 4x4x2 3D torus 25 ns latency per hop 128 GB/s peak bisection bandwidth

TABLE 2. WORKLOAD CONFIGURATIONS.

<i>Online Transaction Processing (TPC-C)</i>	
Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
<i>Web Server</i>	
Apache	16K connections, FastCGI, worker threading model
Zeus	16K connections, FastCGI
<i>Decision Support (TPC-H on DB2)</i>	
Qry 2	Join-dominated, 450 MB buffer pool
Qry 6	Scan-dominated, 450 MB buffer pool
Qry 17	Balanced scan-join, 450 MB buffer pool

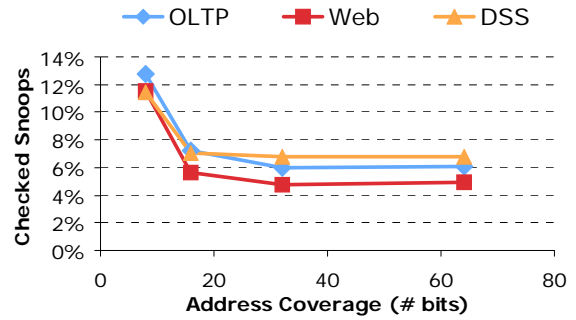
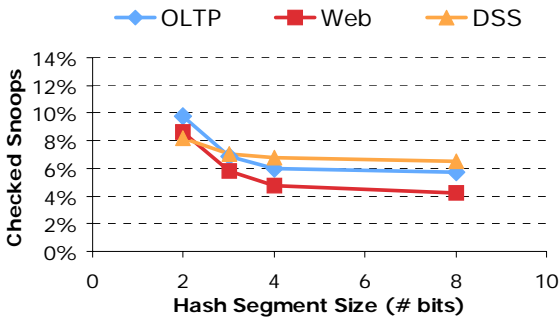


Figure 7. Filter implementation alternatives: (left) hash segment size; (right) address coverage with 4 bit segments.

The filter successfully removes the most significant performance overheads, reducing the average overhead to 4% (peak 9% in OLTP DB2). The two workloads where the filter is least effective are OLTP DB2 and DSS Query 6. We observed that these applications have the most contention for shared data (e.g., races). Moreover, the heavily contended cache blocks in these workloads have latencies in the Tight model that are an order of magnitude higher than the mean request latency (10,000 cycles versus 1200 cycles). This result is indicative of workloads that would benefit from additional tuning; with reduced contention, the effectiveness of the filter will increase and stalls will be further eliminated.

B. Checking Filter

We evaluate the checking filter by first investigating the rate of false positives in a practical implementation. Figure 7 shows two graphs that vary the filter’s hash design with respect to segment size and address coverage (number of segments). Both graphs show the fraction of dirty reads that require checking; hence, lower is better (no filter corresponds to 100% checked). Furthermore, the best coverage (lowest percentage of checked snoops) obtained in both graphs corresponds to the oracle filter results used in Figure 6.

In Figure 7(left), we examine sensitivity to hash segment size. In this graph, we vary the size and number of segments to keep the total address coverage at 32 bits. The segment size has a first-order effect on aliasing in the hash—smaller segments result in more collisions and hence increased false positives. With four bits per segment, the fraction of snoops requiring a check is within 1% of optimal. Going below four bits has an adverse effect on performance.

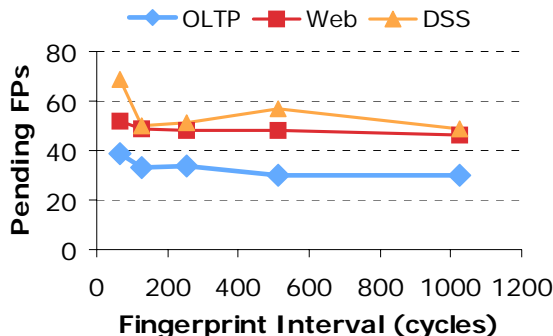


Figure 8. Filter size (FP interval).

In Figure 7(right), we show the complementary result by examining sensitivity to the address coverage. We keep the number of bits per segment fixed at four and vary the number of segments to change the address coverage. With fewer than 32 bits, and particularly with just 8 bits, the aliasing increases to unacceptable levels.

To measure the peak number of in-flight fingerprints, and hence the number of pending hashes required, we varied the rate of periodic fingerprinting. to examine the peak number of in-flight fingerprints, and hence the largest number of pending hashes required. Figure 8 shows that in all cases, no more than 64 pending fingerprints were required. For the workloads we study, total storage for the filter is 1 kB, most of which is kept in a FIFO that does not require fast access.

C. Decoupled Checking

We examine the performance overhead from decoupled checking in Figure 9. This graph shows four experiments: NC models the TRUSS system [7], where no filter is used and checking is coupled to the coherence protocol; ND models decoupled checking without a filter; FC models a system with a filter and coupled checking; finally, FD models both mechanisms proposed here—the checking filter with decoupled checking.

The key result is that the filter effectively removes any sensitivity to the choice of checking protocol. In DSS workloads this is particularly evident. Without a filter, the decoupled protocol adds 5% additional performance overhead; however, with the filter, both coupled and decoupled protocols are under 4% overhead, on average. These results show that the performance impact of decoupled checking is negligible, if the checking filter is used.

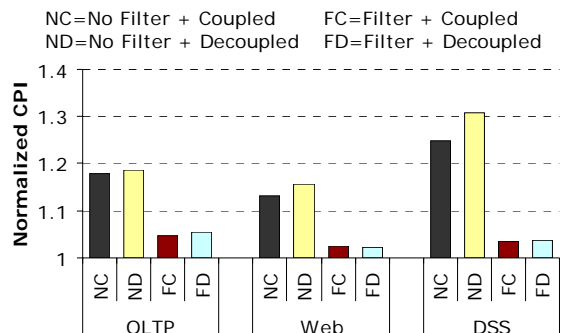


Figure 9. Decoupled Checking.

VII. CONCLUSIONS

To address performance overheads in previous chip-level redundant DSMs, we proposed a checking filter that reduces the number of checking operations impeding the critical path of execution. Furthermore, we propose to decouple checking operations from the coherence protocol, which simplifies the implementation and permits reuse of existing coherence controller hardware. Our simulation results of commercial workloads show average performance overhead is within 4% (9% maximum) of tightly coupled DMR solutions.

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] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, November 1999.
- [2] W. Bartlett and B. Ball. Tandem's approach to fault tolerance. *Tandem Systems Rev.*, 8:84–95, February 1988.
- [3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, July 1970.
- [4] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–17, November-December 2005.
- [5] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. In *IBM Whitepaper*, 1997.
- [6] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design (ICCD)*, pages 522–525, 1992.
- [7] B. T. Gold, J. Kim, J. C. Smolens, E. S. Chung, V. Liaskovitis, E. Nuvitadhi, B. Falsafi, J. C. Hoe, and A. G. Nowatzky. TRUSS: a reliable, scalable server architecture. *IEEE Micro*, 25:51–59, Nov-Dec 2005.
- [8] M. Gomma, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. Int'l Symp. on Computer Architecture (ISCA-30)*, June 2003.
- [9] K. Hardin, J. Fessler, N. Webb, J. Berry, A. Cable, and M. Pulley. Design considerations of phase-locked loop systems for spread spectrum clock generation compatibility. In *Proc. Intl. Symp. on Electromagnetic Compatibility*, 1997.
- [10] R. Kuppuswamy, P. DesRosier, D. Feltham, R. Sheikh, and P. Thadikaran. Full hold-scan systems in microprocessors: Cost/benefit analysis. *Intel Technology Journal*, 8(1), 2004.
- [11] A.-C. Lai and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. of 28th Intl. Symp. on Comp. Arch. (ISCA-28)*, July 2001.
- [12] 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.
- [13] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. Int'l Symp. Computer Architecture (ISCA-29)*, pages 99–110, May 2002.
- [14] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of 7th ACM Intl. Conf. on Management of Data (SIGMOD '88)*, pages 109–116, June 1988.
- [15] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. Int'l Symp. Computer Architecture (ISCA-27)*, June 2000.
- [16] S. R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-accurate deterministic replay for hardware debugging. In *Int'l Conf. Dependable Systems and Networks (DSN)*, June 2006.
- [17] D. P. Sieworek and R. S. S. (Eds.). *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.
- [18] T. J. Slegel, E. Pfeffer, and J. A. Magee. The IBM eserver z990 microprocessor. *IBM Journal of Research and Development*, 48(3), 2004.
- [19] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzky. 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)*, pages 224–234, Oct. 2004.
- [20] J. C. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proc. of 37th IEEE/ACM Intl. Symp. on Microarch. (MICRO 37)*, December 2004.
- [21] D. J. Sorin, M. D. Hill, and D. A. Wood. Dynamic verification of end-to-end multiprocessor invariants. In *Proc. Int'l Conf. Dependable Systems and Networks*, June 2003.
- [22] 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. of 29th Intl. Symp. on Comp. Arch. (ISCA-29)*, June 2002.
- [23] 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.
- [24] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proc. Int'l Symp. Computer Architecture (ISCA-29)*, May 2002.
- [25] 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, Jul-Aug 2006.