TRUSS: A Reliable, Scalable Server Architecture

TRADITIONAL TECHNIQUES THAT MAINFRAMES USE TO INCREASE RELIABILITY—SPECIAL HARDWARE OR CUSTOM SOFTWARE—ARE INCOMPATIBLE WITH COMMODITY SERVER REQUIREMENTS. THE TRUSS ARCHITECTURE PROVIDES RELIABLE, SCALABLE COMPUTATION FOR UNMODIFIED APPLICATION SOFTWARE IN A DISTRIBUTED SHARED-MEMORY MULTIPROCESSOR.

Brian T. Gold
Jangwoo Kim
Jared C. Smolens
Eric S. Chung
Vasileios Liaskovitis
Eriko Nurvitadhi
Babak Falsafi
James C. Hoe
Carnegie Mellon University

Andreas G. Nowatzyk
Cedars-Sinai Medical Center

The day-to-day digital services that users take for granted—from accounting and commercial transactions to residential utilities—often rely on available, reliable information processing and storage. Server reliability is already a critical requirement for e-commerce, where downtime can undercut revenue by as much as $6 million per hour for availability-critical services.[1] Small wonder that reliability has become a key design metric for server platforms.

Unfortunately, although availability and reliability are becoming increasingly crucial, the obstacles to designing, manufacturing, and marketing reliable server platforms are also escalating.[1,2] The gigascale integration trend in semiconductor technology is producing circuits with significant vulnerability to transient error (such as that caused by cosmic radiation) and permanent failure (such as that from device wearout).[3] Reliable mainframe platforms have traditionally used custom components with enhanced reliability, but the cost can be prohibitive.[4] Moreover, these platforms have strong disadvantages: Either they provide a message-passing programming interface, which requires custom software,[5] or they use a small broadcast-based interconnect to share a single physical memory, which compromises their scalability.[6]

In contrast, most modern servers are shared-memory multiprocessors that give programmers the convenience of a global address space. Scalable commodity servers are increasingly based on a cache-coherent distributed shared-memory (DSM) paradigm, which provides excellent scalability while transparently extending the global address space. Unfortunately, DSM servers tend to use potentially unreliable components as building blocks to exploit economies of scale.

The Total Reliability Using Scalable Servers (TRUSS) architecture, developed at Carnegie Mellon, aims to bring reliability to commodity servers. TRUSS features a DSM multiprocessor that incorporates computation and memory storage redundancy to detect and recover from any single point of transient or permanent failure. Because its underlying DSM architecture presents the familiar shared-memory programming model, TRUSS requires no changes to existing applications and only minor modifications to the operating system to support error recovery.

Designing for fault tolerance

Central to TRUSS' practical fault-tolerant design is Membrane, a conceptual fault-isolation boundary that confines the effects of a component failure to the processor, memory, or I/O subsystem in which the failure occurred. With

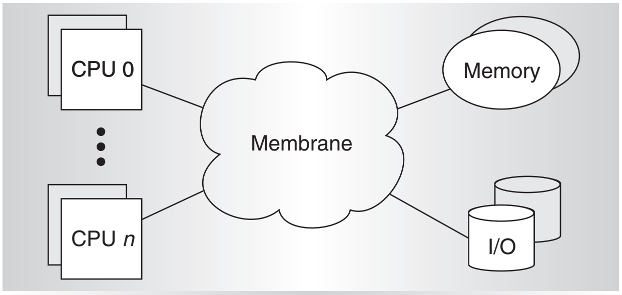

Figure 1. Logical decomposition of error detection and isolation within the TRUSS distributed shared-memory multiprocessor. Each subsystem must detect and recover from an error without involvement from other subsystems. In this manner, each subsystem uses an error detection and recovery scheme optimized for that particular component. The Membrane abstraction composes the various components into a complete fault-tolerant system.

Membrane, each subsystem must individually detect errors and stop them from propagating to the rest of the system. Because the subsystems detect an error locally—before it can spread—the system needs only local recovery to continue correct operation. In essence, the problem of designing a fault-tolerant system becomes a collection of subproblems that are easy to separate and thus more manageable to analyze and solve. We can group processing nodes, for example, in a distributed dual- (DMR) or triple-modular redundant (TMR) scheme to protect against processing errors. The memory subsystem can then rely on both local error correction codes (ECC) and distributed parity for the detection and recovery of errors in memory storage. When we compose the two subsystems through Membrane, the processors on one side simply see the appearance of a reliable memory system on the other. Similarly, traditional redundancy and parity techniques protect storage and other I/O devices without affecting processor or memory design[5,7] (see Figure 1).

To maintain the overall Membrane abstraction, the operation of the interconnection network itself must be error-free and uninterrupted. TRUSS builds on a wealth of prior work in reliable, high-performance interconnects that guarantee packet delivery and guard against data corruption[8,9]—longstanding requirements for high-performance parallel systems.

Two elements are key in enabling TRUSS processors and memory subsystems to satisfy Membrane's requirements for error-free, uninterrupted operation:

- a *master/slave computational redundancy scheme* that protects against processor error or node failure and
- a *distributed-parity memory redundancy scheme* that protects against multibit errors or the complete loss of a node.

In this article, we describe both these elements and the results of a performance evaluation.

Evaluation framework

To evaluate TRUSS performance, we used Flexus, a framework for cycle-accurate full-system simulation of a DSM multiprocessor,[10,11] to simulate a 16-node DSM running Solaris 8. Each node contains a speculative, eight-way out-of-order superscalar processor, a detailed DRAM subsystem model, and an interconnect based on the HP GS1280.[12] We use a wait-free implementation of the total store order (TSO) memory consistency model that enforces memory order at runtime only in the presence of races.[11] Table 1

Table 1. Parameters in the 16-node DSM multiprocessor simulation.

System element	Parameters
Processing nodes	UltraSPARC III instruction set architecture; 4-GHz eight-stage pipeline; out-of-order execution, eight-wide dispatch and retirement; 256-entry reorder buffer
L1 caches	Split instruction and data caches; 64-Kbyte, two-way, two-cycle load-to-use latency; four ports; and 32 miss-status holding registers (MSHRs)
L2 caches	Unified, 8-Mbyte, eight-way, 25-cycle hit latency; one port; and 32 MSHRs
Main memory	60-ns access latency, 32 banks per node, two channels, and 64-byte coherence unit
Protocol controller	1-GHz microcoded controller and 64 transaction contexts
Interconnect	4×4 2D torus, 25-ns latency per hop, 128-GBytes/s peak bisection bandwidth

lists the relevant system parameters.

We evaluated four commercial workloads and three scientific applications:

- *OLTP-DB2* is IBM DB2 version 7.2 enterprise-extended edition running an online transaction processing (OLTP) workload modeled after a 100-warehouse TPC-C installation.
- *OLTP-Oracle* is Oracle Database 10g running the OLTP workload.
- *Web-Apache* is Apache HTTP Server version 2.0 running the SpecWeb99 benchmark.
- *Web-Zeus* is Zeus Web Server version 4.3 running the SpecWeb99 benchmark.
- *Ocean*, *Fast Fourier Transform (FFT)*, and *em3d* are scientific applications that exhibit a range of sharing and network traffic patterns, which we scaled to exceed the system's aggregate cache footprint. In this way, we could evaluate TRUSS under realistic memory-system contention.

Computational redundancy

In TRUSS, processor pairs reside on separate nodes so that the system can tolerate the loss of an entire node. This requirement gives rise to two formidable challenges: coordinating a distributed DMR pair for lockstep operation and having that pair corroborate data as part of error detection and isolation.

Coordination

Given that a variable-latency switching fabric separates processors in a DMR pair, TRUSS requires a coordination mechanism to enforce synchronous lockstep execution in an inherently asynchronous system. Rather than attempting to enforce true simultaneity, we opted for an asymmetric scheme in which the execution of the slave processor in a DMR master-slave pair actually lags behind the master. (The "processor" is the fully deterministic core and caches into which asynchronous inputs—interrupts, cache-line fills, and external coherence requests—feed.) The coordination mechanism enforces the perception of lockstep by replicating at the slave processor the exact sequence and timing of the external, asynchronous inputs that the master processor first observes. Thus, the two processors

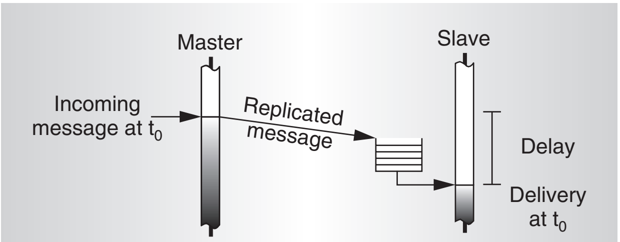

Figure 2. Replicating incoming messages between master and slave. As part of the coordination protocol, the master processor replicates the input and tags it with a delivery timestamp. Both the input and timestamp are forwarded to the slave as a special coordination message. On the slave node, a gated delivery queue presents the forwarded input to the slave processor's interface at precisely the cycle that the timestamp designates (according to a delayed, local time reference).

execute the same instruction sequence despite their physical distance.

Figure 2 illustrates the high-level operation of the master-slave coordination protocol and associated hardware. The coordination mechanism directs all inputs only to the master processor. To ensure that all coordination messages arrive at the slave in time for delivery, the slave runs behind the master at a fixed lag longer than the worst-case transit time for a coordination protocol message. To bound this latency, the master sends coordination messages on the highest-priority channel, and master-slave pairs are neighboring nodes in the network topology.

Because the master and slave nodes have no synchronized clocks, we must be able to modulate their locally synthesized clocks, for example, using down spread-spectrum clock synthesizers.[13] Over time, if the master clock phase drifts too far behind the slave—that is, if coordination protocol messages arrive too close to the required delivery time—the coordination mechanism must actively retard the slave clock until the master clock phase catches up. When the opposite occurs—the slave clock phase drifts too far behind that of the master—the mechanism retards the master clock. This scheme precludes large, unilateral changes to the clock frequency because of thermal throttling or similar mechanisms; rather, such changes must be coordinated between master and slave.

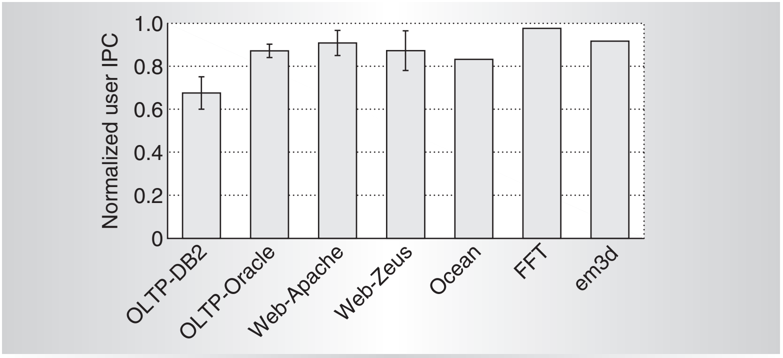

Figure 3. Performance with computational redundancy. We normalize results to a 16-node nonredundant system and show 90 percent confidence intervals on commercial workloads.

Error detection and isolation

Coordination establishes lockstep execution, but TRUSS must also detect and recover from computational errors to satisfy Membrane's requirements. To ensure that no errors leave the logical processor pair, the master and slave must corroborate results and, if error detection reveals an error, recover to a known-good state.

An effective method for processor-side error detection is fingerprinting,[14] which tightly bounds error-detection latency and greatly reduces the required interprocessor communication bandwidth, relative to other detection techniques. Fingerprints compress the execution history of internal processor state into a compact signature, which along with small on-chip checkpointing, provides error detection and recovery.

TRUSS compares fingerprints from the two processors in a lockstep DMR pair. When the master processor generates output data, it communicates first to the slave in a coordination protocol message, which holds a timestamp and a fingerprint summarizing the master computation thus far. At the slave node, the coordination mechanism waits for the slave processor to reach the same execution point. The slave node releases the output data to the rest of the system only if the master and slave fingerprints agree. In this way, the pair corroborates and validates execution correctness up to and including the comparison point.

When the slave detects mismatched fingerprints, indicating an execution error, it restores a checkpoint of architectural state changes to itself and the master, and both resume execution. If the slave node does not detect an error, it discards the previous checkpoint and continues execution. TRUSS recovers from the permanent failure of a master or slave node by either bringing a new master-slave pair online or running the remaining functional node (master or slave) in a nonredundant mode.

TRUSS integrates the error detection and isolation protocol into a three-hop, invalidation-based coherence protocol. In the base protocol, remote nodes forward responses for dirty cache lines directly to the requesting node. TRUSS extends this three-hop forwarding chain to include an additional hop (master to slave) to validate the outbound data. This extra step introduces overhead in any request-reply transaction to a logical processor. For dirty cache reads, for example, the extra step fully manifests in the read transaction's critical path. Outbound data that is not part of a request-reply (writebacks, for example) requires a similar comparison step, but the latency is hidden if no other node is waiting on the writeback result. For operations without irreversible side effects, the master issues the message before it checks the result with the slave.

Because the master accepts request messages while the slave releases replies, a complication arises with network flow control. Guarding against deadlock requires that a node not accept a lower-priority request if back-pressure is blocking it from sending a response on a higher-priority channel. Because the master cannot directly sense back-pressure at the slave's send port, the coordination protocol must keep track of credit and debit counters for the slave's send buffers at the master node. The coordination protocol does not accept an inbound message at the master node unless the counters guarantee that the slave can also absorb the inbound message.

Performance issues

The key performance issues in the coordination protocol are the impact on the round-trip latency of request-reply transactions and network contention from the extra traffic in the master-to-slave channels. Figure 3 shows TRUSS performance relative to a 16-node nonredundant system. For TRUSS, we extended the 4×4 2D torus from the baseline system to a 4×4×2 3D torus, where master and slave nodes are on separate planes of the interconnect topology. This arrangement preserves the master-to-master latency for side-effect-free communications.

In the base system, OLTP-Oracle spends less than 40 percent of execution time in off-chip memory accesses, spending 21 percent of the total time waiting for dirty data. Overhead waiting for this dirty data, along with queuing effects when reading shared data, account for the 15 percent performance penalty in the TRUSS system as compared to the baseline. OLTP-DB2, however, spends 73 percent of execution time on off-chip memory accesses, most of which goes to dirty coherence reads. The additional latency associated with these accesses, coupled with related increases in queuing between master and slave, account for the 35 percent performance penalty in the TRUSS system.

Although both Web-Apache and Web-Zeus spend over 75 percent of execution time in off-chip memory accesses, few of these read modified data from another processor's cache. Moreover, because bandwidth does not generally bound these applications, they can support the additional traffic in the master-to-slave channels. Consequently, Web servers incur a marginal performance penalty in the TRUSS system, relative to the baseline system.

Because their working sets exceed the aggregate cache size, the scientific applications we studied do not spend time on dirty coherence misses and therefore do not incur additional latency from error detection. In ocean and em3d, contention in the master-to-slave channels creates back pressure at the master node and leads to delays on critical-path memory accesses, which accounts for the small performance loss in both applications.

Memory redundancy

TRUSS protects the memory system using Distributed Redundant Memory (Drum), an N+1 distributed-parity scheme akin to redundant arrays of inexpensive disks (RAID) for memory.[7] In this scheme, N data words and one parity word, which the system stores on N+1 different computational nodes, form a parity group, and parity maintenance becomes part of the cache-coherence mechanism.[15] The parity word provides sufficient information to reconstruct any one of the data words within the parity group. Drum complements existing within-node error-protection schemes, such as word- or block-level ECC, and chip-[16,17] or module-level[18] redundancy. It also guards against multibit transient errors—soon reaching prohibitive frequencies in memory[16]—or a single memory component or node failure in a distributed system.

Drum's main goal is to protect memory with little or no performance overhead. As in other distributed parity schemes,[15] Drum relies on ECC to detect multibit errors and does not require a parity lookup upon memory read operations. In the common case of error-free execution, Drum incurs the overhead of updating parity on a memory write operation, such as a cache block writeback.

Contention in distributed parity schemes

A distributed parity scheme can introduce several sources of contention and performance degradation, as Figure 4 shows. Other approaches to distributed parity in a DSM lock the directory entry while the directory controller waits for a parity-update acknowledgement.[15] Generally, acknowledgments help simplify recovery by accounting for all pending parity updates. In workloads such as OLTP, however, which have frequent, simultaneous sharing patterns, concurrent reads stall while the directory entry waits for the parity-update acknowledgment.

A second bottleneck exists at the memory channel and DRAM banks, where program-initiated memory requests and parity updates contend for shared resources. Other proposed techniques[15] uniformly distribute parity information across a node's memory banks. This approach can benefit memory bank load balancing, but for workloads with bursty writeback traffic,[19] parity updates contend with processor requests at the memory channels and increase memory access time.

Finally, parity updates in distributed parity schemes increase the number of network messages. Therefore, in networks with low bisection bandwidth, parity updates can increase network traversal time. However, modern DSMs, such as the HP GS1280,[12] typically use interconnect fabrics designed for worst-case demand, so parity updates in such systems are unlikely to affect message latencies significantly.[15]

Optimizations

Distributed parity in Drum incurs minimal performance overhead through three opti-

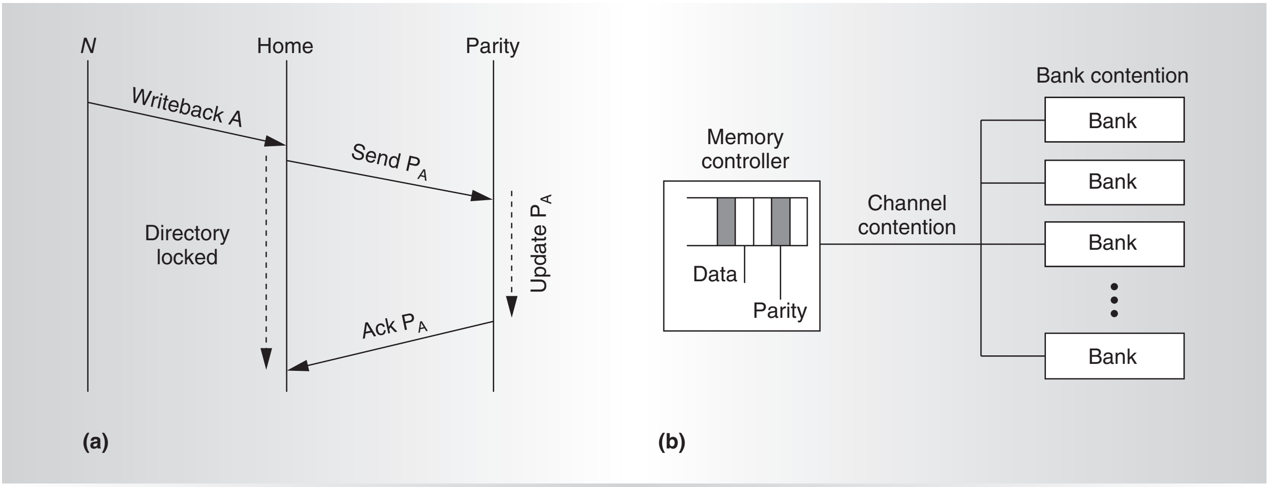

Figure 4. Contention in distributed parity schemes. Directory contention (a) occurs when incoming requests must wait while the directory is locked, stalling critical-path accesses. Memory contention (b) occurs when the addition of parity updates to the memory subsystem stalls critical-path data accesses.

mizations: eliminating parity-update acknowledgments, opting for a lazy scheduling of parity updates, and dedicating memory banks to parity words.

Eliminating parity-update acknowledgments alleviates contention at the directory and reduces the number of network messages, but it can lead to overlapping updates for the same data. Fortunately, parity-update operations are commutative, so performing the updates out of order does not affect parity integrity. For error recovery, however, the system must stop and collect all in-flight messages to guarantee that the memory controllers have completed all updates. The added overhead of quiescing in-flight messages has negligible impact on overall execution time because the system infrequently recovers from multibit errors or hardware failures.

Lazy parity-update scheduling prioritizes data over parity requests at the memory controller, which yields lower memory response time for data requests. Because parity requests are not on the execution's critical path, the memory controller delays them arbitrarily during error-free execution. Drum stores delayed parity requests in a separate parity buffer queue in the memory controller, which identifies and uses idle memory channel and bank cycles for parity requests after servicing bursty data requests. The Drum memory controller also supports the coalescing of parity requests within the parity buffer queue; recomputing the parity effectively coalesces the two requests for the same address, thereby reducing the number of accesses to the parity bank.

To attain higher data throughput, Drum segregates parity values to a few dedicated memory banks, which reduces memory bank contention between data and parity requests, and improves row buffer locality.

Performance

Figure 5 shows the performance of various parity-update organizations, all normalized to a configuration that is not fault-tolerant.

Without prioritized scheduling and dedicated memory banks for parity requests, memory contention causes ocean and FFT to suffer 22 and 18 percent performance losses, respectively. Memory bandwidth bounds performance in both applications, and the applications' footprints exceed the aggregate cache size, which creates significant contention for memory channels and banks. With lazy scheduling and dedicated parity banks, the performance losses for ocean and FFT drop significantly, to 5 and 4 percent. Because writebacks in these applications are bursty, delayed parity requests have ample time to finish.

With parity acknowledgments, directory contention causes OLTP-DB2 and OLTP-Oracle to show 8 and 7 percent performance losses. OLTP exhibits migratory sharing of modified data, where many processors read and

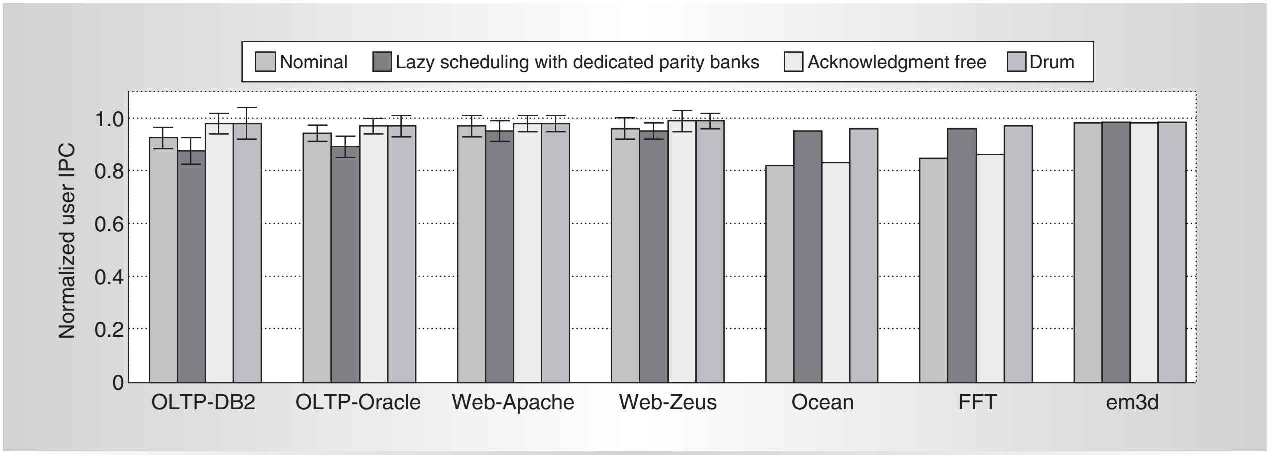

Figure 5. Performance of four parity-update approaches. In the nominal distributed-parity scheme, the directory waits for parity-update acknowledgments and treats all parity updates and data requests equally in the DRAM subsystem. Drum combines the acknowledgment-free and lazy-scheduling-with-dedicated-parity-banks schemes.

write a set of addresses over time. As data blocks pass from one processor's cache to another, outstanding parity updates and acknowledgments delay the release of directory locks. Subsequent reads for these addresses must wait until the parity acknowledgment completes. With the enabling of lazy scheduling and dedicated parity banks, the performance loss grows to 14 percent for OLTP-DB2 and 12 percent for OLTP-Oracle because lazy scheduling further delays parity acknowledgments. However, with no parity acknowledgments, these applications recoup all their performance losses, with or without lazy scheduling and bank dedication. With its combination of acknowledgment-free parity updates, lazy scheduling, and dedicated parity banks, Drum is the only solution that regains the performance losses for all the applications studied. Its performance loss relative to the baseline (non-fault-tolerant) design is only 2 percent on average (at worst 4 percent in ocean).

Reliability and availability will continue to be key design metrics for all future server platforms. The TRUSS server architecture bridges the gap between costly, reliable mainframes and scalable, distributed, shared-memory hardware running commodity application software. Using the Membrane abstraction, TRUSS can confine the effects of a component failure, enabling error detection and recovery schemes optimized for a particular subsystem. MICRO

Acknowledgments
This research was supported in part by NSF awards ACI-0325802 and CCF-0347560, Intel Corp., the Center for Circuit and System Solutions (C2S2), and the Carnegie Mellon CyLab. We thank the SimFlex team at Carnegie Mellon for the simulation infrastructure and valuable feedback on early drafts of this article.

References
1. D. Patterson, keynote address, "Recovery Oriented Computing: A New Research Agenda for a New Century," 2002; http://roc.cs.berkeley.edu/talks/pdf/HPCAkeynote.pdf.
2. J. Hennessy, "The Future of Systems Research," *Computer*, vol. 32, no. 8, Aug. 1999, pp. 27-33.
3. S. Borkar, "Challenges in Reliable System Design in the Presence of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, Nov.-Dec. 2005, pp. 10-16.
4. W. Bartlett and L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, Jan. 2004, pp. 87-96.
5. W. Bartlett and B. Ball, "Tandem's Approach to Fault Tolerance," *Tandem Systems Rev.*, vol. 4, no. 1, Feb. 1998, pp. 84-95.
6. T.J. Slegel, et al., "IBM's S/390 G5 Microprocessor Design," *IEEE Micro*, vol. 19, no. 2, Mar./Apr. 1999, pp. 12-23.
7. D. Patterson, G. Gibson, and R. Katz, "A

Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. Int'l Conf. Management of Data* (SIGMOD-88), ACM Press, 1988, pp. 109-116.

8. J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach,* Morgan Kaufmann, 2003.
9. D.J. Sorin et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery, *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA-02), IEEE CS Press, June 2002, pp. 123-134.
10. N. Hardavellas et al., "Simflex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture," *SIGMETRICS Performance Evaluation Rev.*, vol. 31, no. 4, Apr. 2004, pp. 31-35.
11. T. F. Wenisch et al., "Temporal Streaming of Shared Memory," *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA-05), IEEE CS Press, 2005, pp. 222-233.
12. Z. Cvetanovic, "Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor," *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA-03), IEEE CS Press, 2003, pp. 218-229.
13. K. Hardin et al., "Design Considerations of Phase-Locked Loop Systems for Spread Spectrum Clock Generation Compatibility," *Proc. Int'l Symp. Electromagnetic Compatibility* (EMC-97), IEEE Press, 1997, pp. 302-307.
14. J.C. Smolens et al., "Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS-XI), ACM Press, 2004, pp. 224-234.
15. M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared Memory Multiprocessors," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA-02), IEEE CS Press, 2002, pp. 111-122.
16. T.J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," IBM Corp., 1997.
17. "HP Advanced Memory Protection Technologies," Hewlett Packard, 2003; http://h200001.www2.hp.com/bc/docs/support/SupportManual/c00256943/c00256943.pdf.
18. "Hot Plug Raid Memory Technology for Fault Tolerance and Scalability," HP white paper, Hewlett Packard, 2003. http://h200001.www2.hp.com/bc/docs/support/SupportManual/c00257001/c00257001.pdf.
19. S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture* (ISCA-95), IEEE CS Press, 1995, pp. 24-36.

Brian T. Gold is a PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include reliable computer systems and parallel computer architectures. Gold has an MS in computer engineering from Virginia Tech. He is a student member of the IEEE and ACM.

Jangwoo Kim is a PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include reliable server architecture and full system simulation. Kim has an MEng in computer science from Cornell University. He is a student member of the IEEE.

Jared C. Smolens is a PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include microarchitecture, multiprocessor architecture, and performance modeling. Smolens has an MS in electrical and computer engineering from Carnegie Mellon University. He is a student member of the IEEE.

Eric S. Chung is a PhD student in electrical and computer engineering at Carnegie Mellon University. His research interests include designing and prototyping scalable, reliable server architectures and transactional memory. Chung has a BS in electrical and computer engineering from the University of California at Berkeley. He is a student member of the IEEE and ACM.

Vasileios Liaskovitis is an MS student in electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture and algorithms for pattern recognition. Liaskovitis has a BS in electrical and computer engineering

from the National Technical University of Athens, Greece. He is a student member of IEEE and ACM.

Eriko Nurvitadhi is a PhD student in electrical and computer engineering at Carnegie Mellon University. He received an MS in computer engineering from Oregon State University. His research interests are in computer architecture, including prototyping and transactional memory. He is a student member of the IEEE and ACM.

Babak Falsafi is an associate professor of electrical and computer engineering and Sloan Research Fellow at Carnegie Mellon University. His research interests include computer architecture with emphasis on high-performance memory systems, nanoscale CMOS architecture, and tools to evaluate computer system performance. Falsafi has a PhD in computer science from the University of Wisconsin and is a member of the IEEE and ACM.

James C. Hoe is an associate professor of electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture and high-level hardware description and synthesis. Hoe has a PhD in electrical engineering and computer science from MIT. He is a member of the IEEE and ACM.

Andreas G. Nowatzyk is the associate director of the Minimally Invasive Surgery Technology Institute at the Cedars-Sinai Medical Center, where he works on highly reliable, high-performance computer systems that process real-time image data in operating rooms. Nowatzyk has a PhD in computer science from Carnegie Mellon University. He is a member of the IEEE and ACM.

Direct questions and comments about this article to Babak Falsafi, Electrical and Computer Engineering Dept., Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213; babak@ece.cmu.edu.