

---

# A HARDWARE MEMORY RACE RECORDER FOR DETERMINISTIC REPLAY

---

THE FLIGHT DATA RECORDER CONTINUALLY LOGS MEMORY RACES IN A MULTITHREADED EXECUTION, ENABLING THE DETERMINISTIC REPLAY INVALUABLE FOR DEBUGGING CONCURRENCY ERRORS, YET ADDS ONLY MODEST HARDWARE TO A MULTICORE CHIP. IN EXPERIMENTS, RECORDING INCURRED LESS THAN 2 PERCENT RUNTIME OVERHEAD.

..... As hardware vendors transition to multicore chips, software vendors face increased software reliability challenges. To effectively debug software in this new world, developers must be able to replay executions that exhibit a bug so that they can zero in on concurrency bugs—especially intermittent ones. Such deterministic replay also aids fault detection and recovery, intrusion detection, and the like.

Unfortunately, in multicore chips, neither the software nor the hardware environments provide a practical replayer. This starts an unhealthy downward spiral. If programmers have insufficient tools with which to write and debug multithreaded applications for such chips, performance gains stall. Vendors could end up delivering buggy or late software or both. If users fail to see effective computer performance gains, they might not be so eager to invest in new machines, and vendor revenues could flatten.

To stop or even reverse this spiral, we advocate augmenting multicore chips with the Flight Data Recorder. Like an aircraft flight data recorder, FDR records information at low overhead during normal operation. To enable deterministic replay in multicore systems, the most critical problem that FDR must solve is memory race recording. To do so, it must log sufficient

information to order the outcomes of all conflicting memory accesses.

FDR efficiently solves this problem for multicore designs that use sequential consistency *or* total store order (x86-like) with modest hardware changes.<sup>1-3</sup> Principally, it augments each core with a dynamic instruction counter and a small local memory for logical time stamps, which are taken from instruction counts, and it piggybacks time stamps on some coherence messages to order conflicting accesses.

In viability tests, FDR was able to record memory races for billions of instructions with only megabytes of log storage—on average, 1 byte for every thousand instructions executed. Memory race recording incurred less than 2 percent runtime overhead and added about 10 percent interconnect bandwidth overhead. These results support FDR as a practical option for recording races in future multicore systems.

## Operational context

Deterministic replay requires both a recorder and a replayer. The *recorder* logs information during multithreaded program execution that is sufficient to enable deterministic replay. In the postmortem analysis, the *replayer* uses the logged information, together with the program binary, to faithfully replay the original

Min Xu  
VMware

Rastislav Bodík  
University of California,  
Berkeley

Mark D. Hill  
University of Wisconsin  
at Madison

execution. The replay will always exercise the same bugs and produce the same outputs.

Ideally, the recorder lets the program run at nearly full speed so that it can always be left on. This in turn facilitates the post-mortem analysis of bugs recorded in the field, rather than only those observed in the lab, in much the same way that a flight data recorder operates on an aircraft.

The recorder must perform three tasks, the first of which is to record initial state. This task is trivial if recording begins when a program starts execution. If not, the problem reduces to taking a checkpoint, and checkpointing is a well-studied problem with many old (copy-on-write pages) and new solutions (logging selected read values<sup>4</sup>). With sufficiently long replay intervals, even high checkpoint overhead will affect performance only minimally.

The recorder's second task is to log inputs from outside the system being recorded.<sup>4</sup> An input can include both a value and a time stamp. Input values include program reads of devices (I/O space), values copied into memory by devices with direct memory access, and values that affect interrupts. Input time stamps record the I/O timing, for example, the dynamic instruction count when a processor services an interrupt.

Finally, the recorder must log the outcome of all memory conflicts (or races), so that the replayer can order the memory accesses as in the recorded execution. Two memory accesses conflict if they are from different threads, access same memory location, and at least one of them is a write.

Although there are practical solutions for the first two problems, existing designs for memory race recorders have highly complex hardware and significant runtime overhead. Practical memory race recording thus holds the key to making recorders viable. The FDR algorithm we have developed focuses on the memory race recording problem and adopts existing solutions for the first two problems.

At VMworld 2006, Mendel Rosenblum of VMware demonstrated an instruction-exact, record-replay prototype (not a screen-shot movie recorder), which recorded

a colleague painting a picture with Microsoft Paint and then repainted that picture in replay mode. In our judgment, such a demonstration with a multithreaded program will require hardware race recorders, because existing software race recorders either significantly slow program execution<sup>5,6</sup> or have hard-to-meet requirements, such as disallowing data races in the (buggy) program or requiring it to run on a uni-processor system.

## Multicore chip design modifications

In creating a memory race recorder for multiprocessors that implement sequential consistency or total store order, the basic idea is to first have each core assign a count (time stamp) to the instructions it executes and then remember the time stamp or an approximation of when it last accessed each memory block. When core  $C_2$  seeks to access a block that core  $C_1$  accessed,  $C_1$ 's coherence response includes  $C_1$ 's time stamp for the block. Thus,  $C_2$  can log an entry ordering its conflicting access after  $C_1$ 's access. Optimizations enable FDR to elide logging on most coherence responses and to store only a few time stamps.

Figure 1 shows the multicore design we assume before and after adding our race recorder. The system includes dynamic RAM (DRAM) and I/O bridges attached to a single multicore chip. The multicore chip includes two or more single-threaded processor cores, private write-back L1 instruction and data caches, a shared banked L2 cache, and a point-to-point interconnect. A MOSI (modified-owned-shared-invalid) directory protocol keeps caches coherent with directory entries at L2 cache banks, and L1 caches do not notify the directory on shared replacements.

As the shaded part of Figure 1 shows, we add a local dynamic instruction counter (IC) to each core that assigns a logical time stamp to each instruction that the core commits. Each core also includes a time stamp memory (TSM) of 24 Kbytes (a modest size). A TSM caches the time stamps of recently accessed memory blocks, evicting older time stamps to accommodate newer ones. For each memory block, a TSM can provide a time stamp greater than or

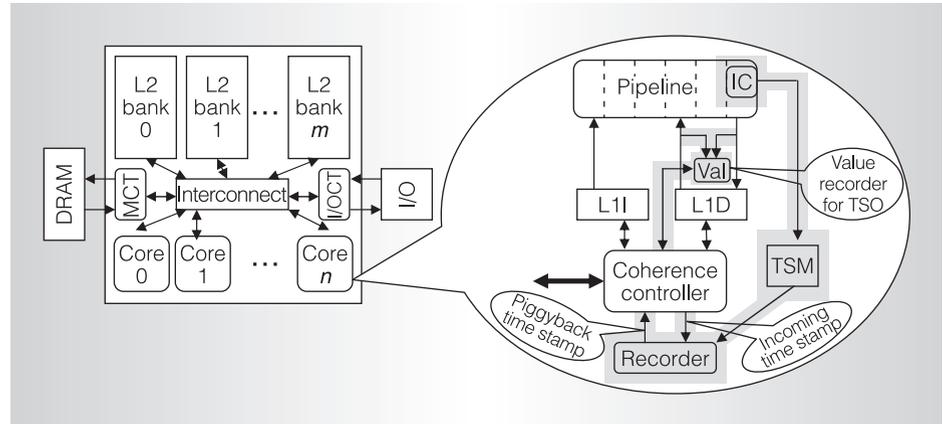


Figure 1. A base multicore system (unshaded) supplemented with FDR's hardware for memory race recording (shaded).

equal to when the local core last accessed the block. The TSM can safely approximate the time stamp for block B not found in the TSM with the oldest time stamp in block B's set. TSMs must use true least recently used eviction, but can use any associativity.

FDR also adds a piggybacked time stamp in the coherence protocol response messages, such as data and invalidation acknowledgments. The piggybacked time stamp is a message payload that does not affect the coherence protocol, but the receiving core can log this time stamp to order conflicting accesses.

Although Figure 1 does not show it, each core includes logging hardware to record selected time stamps and core identifiers. A simple implementation writes the log to the L2 cache for physical (or virtual) memory reserved for each core.

An optional part of the race recorder is support for total store order, which requires that each core include simple, local logic to detect when a read might violate sequential consistency so that the core can log that value.

### How race recording works

Figure 2 illustrates how race recording works in FDR. Figure 2a depicts the code of two threads,  $i$  and  $j$ , and Figure 2b shows the sequence of FDR actions for memory location B. For simplicity, we assume that location B is the entire memory block B and that threads  $i$  and  $j$  run on cores  $C_1$  and  $C_2$ ,

respectively. After thread  $i$  finishes its instruction 2 (denoted  $i:2$ ), we assume that its cache contains block B in the modified (M) state and  $C_2$ 's copy is invalid (I).

The numbers in Figure 2b correspond to the following events:

1. As  $C_1$  writes memory block B,  $C_1$  records its current instruction count of 2 in its TSM as block B's time stamp.
2. When  $C_2$  seeks to write block B at instruction count 3, an L1 miss occurs.
3.  $C_2$  sends an exclusive coherence request (GETX) to the L2 directory.
4. The directory forwards the request to  $C_1$ .
5.  $C_1$  looks up block B's time stamp of 2 from its TSM and responds to  $C_2$  with the data block B and a piggybacked time stamp.

The last action (not shown) is that  $C_2$  writes a " $i:2$  is before  $j:3$ " record into  $j$ 's log and completes its instruction 3, writing memory location B. A replayer can use this recorded information to replay the race for memory location B in the same order. A replay of thread  $j$ , for example, reads the log entry, executes thread  $j$ 's instructions to  $j:2$  (just before  $j:3$ ), waits for thread  $i$  to execute past  $i:2$ , and then executes  $j:3$  and subsequent thread  $j$  instructions until the replay stalls at thread  $j$ 's next log entry.

As we describe later, FDR need not record the arc for memory location A from

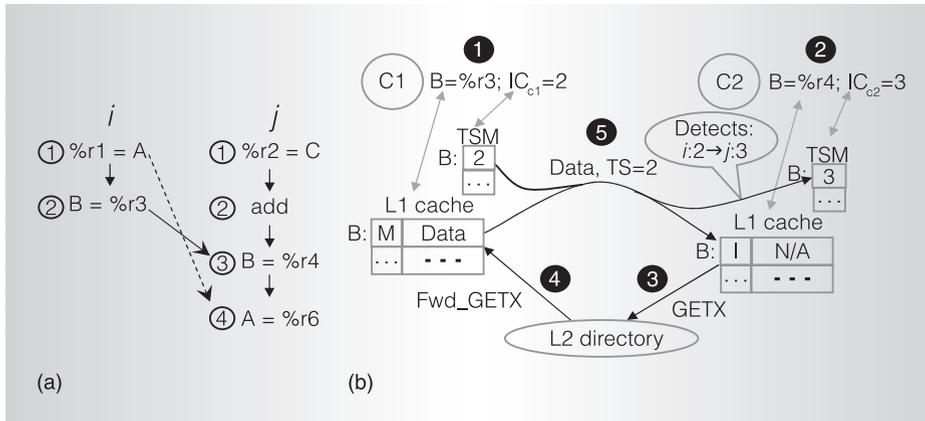


Figure 2. How race recording works in FDR. A multithreaded execution (a) and how FDR records a shared memory race for that execution (b).

$i:1$  to  $j:4$  (dashed line in Figure 2a). The replayer can infer this arc, because it knows that  $i:1$  is before  $i:2$  ( $i$ 's program order),  $i:2$  is before  $j:3$  (logged), and  $j:3$  is before  $j:4$  ( $j$ 's program order). It also knows that the relation “happens-before” is transitive: The relations “ $x$  is before  $y$ ” and “ $y$  is before  $z$ ” imply the relation “ $x$  is before  $z$ .”

Following similar reasoning, FDR's TSMs can be caches that don't explicitly store time stamps for all memory blocks. For example, suppose that thread  $i$  accesses memory location B at instruction count 1000 and executes long enough for  $i$ 's TSM to evict B's time stamp. Much later, if thread  $j$  accesses B at  $IC_j$ , then thread  $j$  may log “ $i:2000$  is before  $j:IC_j$ ,” where 2000 is provided by  $i$ 's TSM. This log entry is sufficient, because by transitivity, the relations “ $i:1000$  is before  $i:2000$ ” ( $i$ 's program order) and “ $i:2000$  is before  $j:IC_j$ ” (logged) imply the relation “ $i:1000$  is before  $j:IC_j$ .”

We also assume that it is not acceptable to store time stamps in the shared L2 cache or to change the DRAM. Instead, with modest coherence protocol changes, the protocol recovers missing memory orderings. After  $C_2$  performs an L1 write-back of B, for example, the L2 directory entry for B continues to remember  $C_2$ . To accommodate  $C_1$ 's subsequent request for block B, the coherence protocol is modified to ask  $C_2$  to query its TSM for block B's time stamp or a larger approximation.

### Order of coherence operations

FDR records the order of coherence operations so that the replayer can replay them in the same order: The same coherence order yields the same execution. Some recorders assume a snooping system and record the total order of bus activity.<sup>7</sup> In contrast, FDR records only a partial order of coherence activity, recording the order of conflicting accesses but not necessarily that of nonconflicting accesses. Although it seems intuitive that deterministic replay is possible without ordering nonconflicting accesses, proof of that requires more formalism.

One formal proof uses sequential consistency and the notion of equivalent sequentially consistent executions. Sequential consistency semantics require that the memory accesses of all cores appear to be interleaved into a single total order. Two executions are equivalent if every read obtains the same value and the final memory state is the same. The proof's author, Robert Netzer, showed that recording the conflicting accesses of a sequentially consistent execution, as FDR does, is sufficient to replay an equivalent sequentially consistent execution.

### Controlling log growth

FDR would not be viable if it created log entries on every interaction among cores, which would require considerable write bandwidth and generate large logs. For

a fixed log size, dividing log growth rate by a factor  $R$  is the same as multiplying the supported recording length by the same factor.

FDR optimizes log growth in two ways. First, it simplifies Netzer's transitive reduction optimization for hardware implementation.<sup>1</sup> In Figure 2a, transitive reduction freed FDR from recording the dashed arc at memory location A because the arc was implied both by transitivity through program order and by the recorded arc for memory location B.

Overall, transitive reduction can reduce log size growth 10- to 1,000-fold. Threads often run independently for many instructions, and when they do interact, they frequently use a single synchronization variable to coordinate multiple data accesses. Suppose, for example, that one thread accesses data blocks  $D_1$ ,  $D_2$ , and  $D_3$  and then releases lock  $L$  while another thread waits for  $L$  to be freed, acquires it, and then accesses  $D_1$ ,  $D_2$ , and  $D_3$ . With transitive reduction, FDR will log an entry only for the block containing  $L$ , even though FDR is ignorant of program semantics, such as where locks are or what they mean.

Hardware for implementing transitive reduction is local and simple. Each core  $C_j$  records a table of the largest time stamp it received from each other core  $C_i$ .  $C_j$  does not log an entry from  $C_i$  with  $IC_i$  if the time stamp it has stored for  $C_i$  is greater than or equal to the newly received  $IC_i$ .

FDR improves on Netzer's original transitive reduction in three ways: It uses TSMs to approximate missing time stamps; tracks scalar, not vector, time stamps; and provides a hardware implementation.

FDR also uses an optional regulated transitive reduction optimization to further reduce log size by as much as 30 percent.<sup>2,3</sup> This optimization often records stricter dependencies than necessary for replay, both to allow more log entries to be removed through transitive reduction and to compress entries with a process somewhat analogous to vectorization.

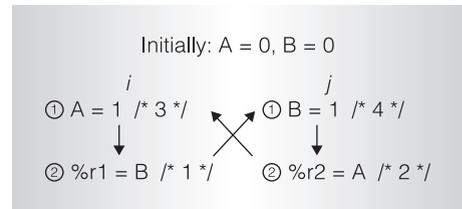


Figure 3. A sample total store order execution that sequential consistency disallows because the write buffer delays the writes. The numbers in /\* \*/ denote the memory ordering.

### Recording total store order executions

Previous race recorders, whether implemented in hardware or software, support sequential consistency only, which makes them unusable by most deployed hardware. To address this problem, we created an FDR extension to support total store order in addition to sequential consistency. Total store order is well defined and likely to be implemented by many x86 systems as a valid realization of processor consistency.

With this extension, FDR can record total store order executions without forcing the executions to conform to sequential consistency and only modestly increasing log size and hardware complexity.

Total store order creates challenges for race recording because it relaxes write-to-read ordering to the shared memory. Informally, a core can implement a hardware first-in-first-out write buffer. Figure 3 shows a total store order execution that sequential consistency disallows. In this execution, thread  $i$  first writes memory location A and then reads a different memory location B, and thread  $j$  first writes B and then reads A. Because of the write buffers, the ordering of the two reads occurs before that of the writes. For this execution, a race recorder that assumes sequential consistency would log two interthread arcs " $i:2$  is before  $j:1$ " and " $j:2$  is before  $i:1$ ." During the replay, if the replayer follows the sequential consistency order, the replay will deadlock because of the cycle of dependencies formed by the recorded arcs and

program order arcs “ $i:1$  is before  $i:2$ ” and “ $j:1$  is before  $j:2$ .”

To handle both sequential consistency and total store order executions, we created a hybrid *order-value* recorder<sup>3</sup> that detects and reacts to problematic reads, such as those that cause replay deadlocks because they are ordered (at memory) before writes that are earlier in the program order.

The hybrid recorder detects read  $R$  at  $IC_n$  as problematic if it obtains a value  $V$  from the cache, while one or more earlier writes  $W_0, \dots, W_{n-1}$  (where  $ICs$  are less than  $IC_n$ ) are still in the write buffer and the cache block containing  $V$  is invalidated before all writes  $W_0, \dots, W_{n-1}$  exit the write buffer. The recorder reacts to a problematic read  $R$  by logging its instruction count  $IC_n$  and value read  $V$ . With this information, a replayer can subsequently replay problematic reads by value, rather than by ordering.

Our logic for detecting problematic reads is similar to that used in aggressive implementations of sequential consistency,<sup>8</sup> but our reaction logs a tuple and does not trigger a misspeculation recovery to obtain sequential consistency compliance. Logging problematic read values only slightly increases log size because reads that could violate sequential consistency generally occur infrequently.

## Evaluation methods and results

To evaluate FDR’s memory race recording, we used a full-system simulation running on the Wisconsin General Execution-driven Multiprocessor Simulator (GEMS; <http://www.cs.wisc.edu/gems>), which augments the Virtutech Simics simulator. GEMS models a Sparc multicore system in sufficient detail to run the unmodified Solaris 9 operating system. The target system has one multicore chip with four cores; a 15-cycle, shared 16-Mbyte L2 cache; and an 80-ns off-chip DRAM. Each core is 1 GHz with 64-Kbyte L1 caches that are two-way-issue, in-order with a single-cycle split instruction and data. Each core’s TSM is 24 Kbytes and holds 2,048 time stamps.

We exercised the system with four commercial workloads:

- *Apache* is a static Web-serving workload.
- *Online transaction processing* models database activities of a wholesale supplier, with many concurrent users performing transactions.
- *JBB* is a server-side Java benchmark that models a three-tier system, focusing on the middleware server business logic.
- *Zeus* is another static Web-serving workload that seeks to improve performance by using a fixed number of single-thread I/O processes to handle Web requests.

Figure 4 displays key results. As Figure 4a shows, FDR’s memory race log grows about 1 byte per thousand instructions executed, which means that it can record memory races for billions of instructions with only megabytes of log storage. Figures 4b and 4c show that race recording has negligible runtime overhead and adds about 10 percent interconnect bandwidth overhead, which is tolerable.

Our experiments with FDR race recording made many assumptions about the base system, such as having a single multicore system, single-threaded cores, directory coherence, write-back caches, and reliance on sequential or total store order consistency. Future work should focus on relaxing some of these system assumptions.

Some FDR extensions are likely to be trivial. Providing support to systems with more than one multicore chip depends on the coherence protocol. It is straightforward to handle multithreaded cores, as in hyper-threading, by augmenting the now-shared L1 caches with state bits to trigger pseudo-coherence logging events when different threads of the same core interact.

FDR currently records time stamps of evicted blocks with modest coherence protocol changes. A potentially better alternative is to leave the protocol unchanged by storing at each L2 cache bank the time stamps for each core’s last block write-back.

Other extensions require more creativity. Race recorders for systems with snooping rather than directory coherence could require other methods of reducing race logs, such as

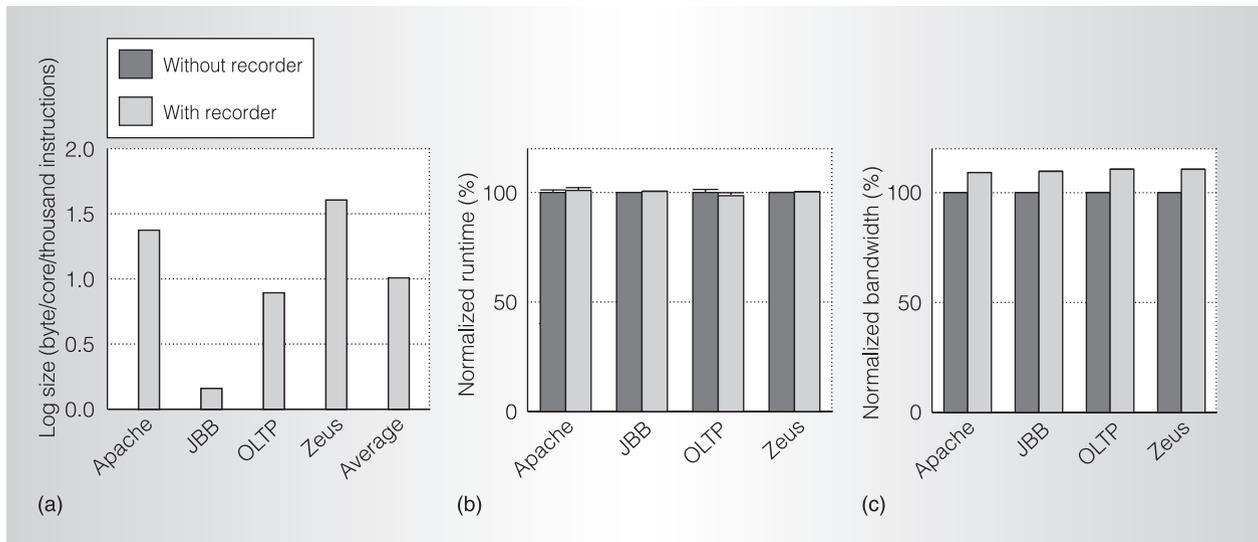


Figure 4. Results of evaluating FDR on the Wisconsin Multifacet GEMS. FDR's log size growth rate (a), runtime overhead (b), and interconnect bandwidth overhead (c).

storing selected written-back time stamps as just described, or dividing memory accesses into temporal strata.<sup>9</sup> Moreover, although handling write-through caches to a private write-back cache appears easy, recording a system with write-through to a single shared cache might require additional changes to the coherence protocol.

Finally, at least one extension has no known efficient solution: race recording with memory consistency models that are more relaxed than total store order. In this case, our order-value hybrid does not directly apply.

Clearly, an investment in modest chip resources has great potential to ease the challenges of debugging the multithreaded software that both users and vendors will continue to demand.

MICRO

### Acknowledgments

We thank Daniel Gibson and Michael Swift, as well as the people and groups we acknowledged in our previous accounts of FDR.

This work is supported in part by National Science Foundation grants CCF-0085949, CCR-0093275, CCR-0105721, EIA/CNS-0103670, CCR-0105721, EIA/CNS-0205286, CNS-0225610, CCR-0243657, CCR-0324878,

and CCR-0326577; by the Defense Advanced Research Projects Agency under contract NBCHC020056; and through awards from UC Micro and Okawa Foundation and donations from IBM, Intel, Microsoft, and Sun.

The views expressed herein are not necessarily those of DARPA, IBM, Intel, Microsoft, NSF, Sun, or VMware. Xu performed this work as a PhD student at the University of Wisconsin, Madison, before joining VMware. Hill has a significant financial interest in Sun Microsystems.

### References

1. M. Xu, R. Bodík, and M.D. Hill, "A 'Flight Data Recorder' for Enabling Full-System Multiprocessor Deterministic Replay," *Proc. 30th Ann. Int'l Symp. Computer Architecture (ISCA 03)*, IEEE CS Press, 2003, pp. 122-133.
2. M. Xu, R. Bodík, and M.D. Hill, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," *Proc. 12th Int'l Conf. Architectural Support Programming Languages and Operating Systems (ASPLOS 06)*, ACM Press, 2006, pp. 49-60.
3. M. Xu, "Race Recording for Multithreaded Deterministic Replay Using Multiprocessor Hardware," doctoral dissertation, CS Dept., University of Wisconsin—Madison, 2006.

4. S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA 05)*, IEEE CS Press, 2005, pp. 284-295.
5. T.J. Leblanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant-Replay," *IEEE Trans. Computers*, vol. C-36, no. 4, Apr. 1987, pp. 471-482.
6. R.H.B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging (PADD 93)*, ACM Press, 1993, pp. 1-11.
7. D.F. Bacon and S.C. Goldstein, "Hardware-Assisted Replay of Multiprocessor Programs," *ACM SIGPLAN Notices: Proc. ACM/ONR Workshop Parallel and Distributed Debugging*, ACM Press, 1991, pp. 194-206.
8. K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. Int'l Conf. Parallel Processing*, vol. 1, CRC Press, 1991, pp. 355-364.
9. S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," *Proc. 12th Int'l Conf. Architectural Support Programming Languages and Operating Systems (ASPLOS 06)*, ACM Press, 2006, pp. 229-240.

**Min Xu** is a member of the technical staff at VMware. His interests include hardware and software techniques in deterministic

replay. Xu has a PhD in electrical engineering from the University of Wisconsin—Madison. He is a member of the ACM and IEEE.

**Rastislav Bodík** is an assistant professor in the Computer Sciences Division at the University of California, Berkeley. His research interests include static and dynamic program analysis, software tools, and compilation. He has a PhD in computer science from the University of Pittsburgh.

**Mark D. Hill** is a professor in both the Computer Sciences and Electrical and Computer Engineering Departments at the University of Wisconsin—Madison. His research interests include parallel computer system design, memory system design, and computer simulation. He has a PhD in computer science from the University of California, Berkeley, and is a fellow of the IEEE and ACM.

Direct questions and comments about this article to Mark D. Hill, Computer Sciences Dept., University of Wisconsin—Madison, 1210 West Dayton St., Madison, WI 53706-1685; markhill@cs.wisc.edu.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.