An Improved "Flight Data Recorder" of Shared Memory Races

Min Xu
VMware Inc.

Rastislav Bodík
University of California, Berkeley

Mark D. Hill
University of Wisconsin-Madison

ADDING SIMPLE RECORDER HARDWARE TO A MULTICORE CHIP CAN ENABLE DETERMINISTIC REPLAY FOR CYCLIC DEBUGGING.

Hardware vendors are currently transitioning from single-threaded microprocessors to chips that integrate multiple processor cores and threads in what is variously called *multicore, chip multiprocessing,* and *chip multithreading*.

At the same time, developing and maintaining reliable software continues to challenge software vendors. Effectively debugging the software is critical. Valuable to any debugger, *deterministic replay* enables a developer to re-execute the (buggy) program and zero in on bugs that faithfully re-appear. Moreover, deterministic replay can also be useful for fault detection/recovery, intrusion detection, and other applications.

To effectively use multicores, however, programmers must write and debug *multithreaded* applications. Unfortunately, neither the software or hardware environments of multicores provide deterministic replay. If nothing is done, software could be delivered later, with more bugs, or both. If users fail to see continued doubling of effective computer performance, vendor profits could be negatively affected.

To mitigate some of the gloom in this situation, we advocate augmenting multicores with modest hardware, called a **Flight Data Recorder (FDR)**, that records sufficient information in a log to enable deterministic replay. Like an aircraft flight data recorder, FDR continually records information at low overhead during normal operation.

This paper provides a gentle introduction to how FDR solves the critical problem of *recording memory races*, while our previous work discusses all aspects, including providing algorithms and related work [9, 10, 8]. A race recorder must log sufficient information to order the outcomes of all conflicting memory accesses. Two accesses (reads or writes) *conflict* if they are from different threads, access the same memory block, and at least one of them is a write. In particular, FDR's race recording:

- supports multicore designs using sequential consistency (SC) or total store order (TSO), which is x86-like;
- augments each core with a dynamic instruction counter and a small local memory for logical timestamps (which are derived from instruction counts);
- piggybacks timestamps on some coherence messages; and
- exploits transitivity to log about one byte per thousand instructions executed.

Memory Race Recording Context

Before we develop FDR, we summarize the context in which it operates. We assume a *recorder* and a *replayer.*

The *recorder* logs information during multithreaded program execution that is sufficient to enable deterministic replay. Ideally, it should allow program to run at (nearly) full speed so the recorder can be "always on" for post-mortem analysis of bugs in the field (like aircraft flight data recorders).

The recorder must solve three sub-problems. First, it must record *initial state*. This is trivially done if recording begins as a program starts execution. Otherwise, the problem reduces to taking a checkpoint. Checkpointing is a well-studied problem with many old (e.g., copy-on-write pages) and new solutions (logging selected read values [5]). With sufficiently long replay intervals, even high checkpoint overhead will only minimally affect performance.

Second, a recorder must log *inputs* from outside the system being recorded, which is also a well-studied problem. An input may include both a value and a timestamp. Input values include program reads of devices (I/O space), values copied into memory by devices with direct memory access (DMA), and values that affect (e.g., vector) interrupts. Input timestamps record the I/O "timing" (e.g., the dynamic instruction count when a processor services an interrupt).

Third, the recorder must log the outcome of all *memory races*, which is the subject of this paper.

After recording, the *replayer* uses the logged information, together with the program binary, to faithfully replay the original execution. The replay will always exercise the same bug(s) and produce the same output(s). In many cases, replayer execution time is less critical, because it is used only when bugs are discovered, which is (hopefully) a small fraction of all executions. Thus, it may be sufficient to do replay with a software simulator. We have only developed a simple replayer for testing recorders.

Memory race recording holds the key for future recorders for multithreaded programs on multicore systems. At VMworld 2006, Mendel Rosenblum demonstrated a record-replay prototype where a colleague painted a picture with Microsoft Paint and replay re-painted it. They received a rousing ovation. In the future, we would like to do such a demonstration with a multithreaded program. In our judgement (prior to Xu joining VMware), doing so will *require hardware memory race recording*. This is because existing software race recorders slow down program

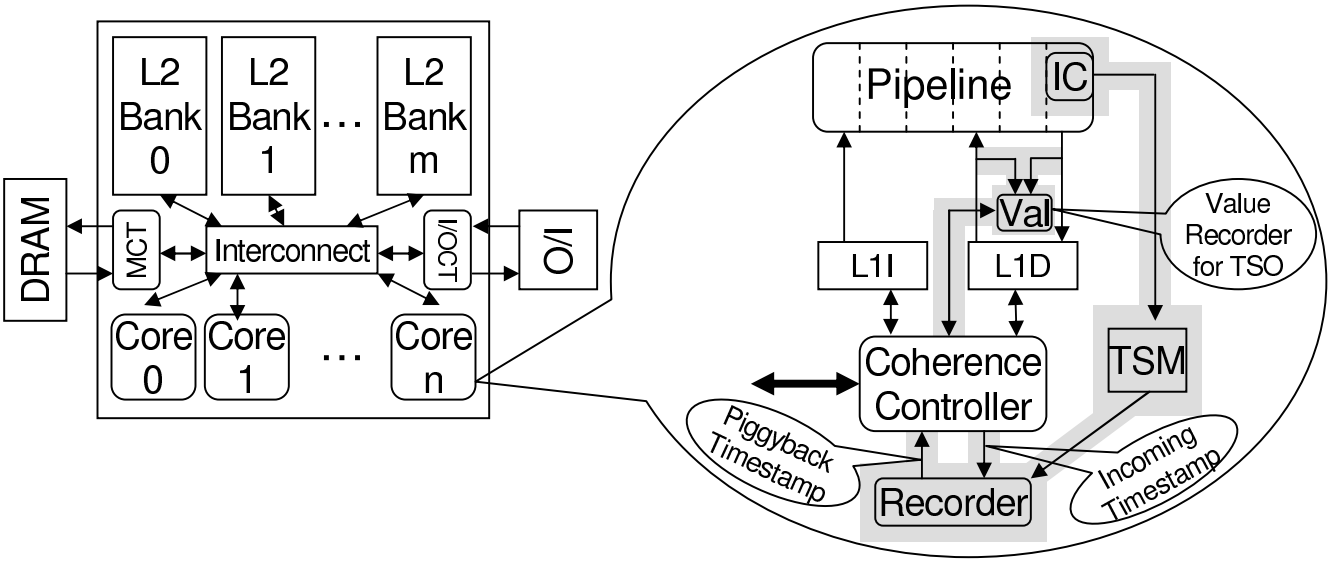

Figure 1: A base multicore (unshaded) supplemented with FDR memory race recording hardware (shaded)

execution tremendously [3, 6], or have hard-to-meet requirements (*i.e,.* (buggy) programs never have data races and/or programs always run on a uniprocessor system).

FDR Hardware Memory Race Recorder

This section develops the FDR memory race recorder for multiprocessors that implement SC or TSO. The basic idea has two components. First, we have each core assign a count (timestamp) to the instructions it executes and remember (an approximation of) the timestamp when it last accessed each memory block. When core C2 seeks to access a block accessed by core C1, core C1's coherence response includes the core C1's timestamp for the block, so that core C2 can log an entry ordering its conflicting access after the core C1's access. Optimizations enable FDR to elide logging on most coherence responses and to store few timestamps.

To be concrete, we assume the base multicore design depicted in Figure 1 (unshaded). Our system includes DRAM and I/O bridges attached to a single multicore chip. The multicore includes a two or more single-threaded processor cores, private writeback L1 instruction and data caches, a shared banked L2 cache, and a point-to-point interconnect. Caches are kept coherent with a MOSI directory protocol (e.g., directory entries at L2 cache banks) where L1 caches do not notify the directory on shared replacements. Later we touch upon issues relaxing these concrete assumptions.

We now discuss FDR's added hardware (shaded in Figure 1) and operation.

Added Hardware. FDR makes a few hardware changes. First, FDR adds a local *dynamic instruction counter (IC)* to each core that assigns a logical *timestamp* to each instruction that the core commits.

Second, each core includes a *timestamp memory (TSM)* of modest size (e.g., our evaluation uses a 24KB TSM). For each memory block, a TSM can provide a timestamp greater than or equal to when the local core last accessed the block. A TSM caches the timestamps of recently-accessed memory blocks. It evicts older timestamps to accommodate the newer ones. The TSM can safely approximate the timestamp for block B not found in the TSM with the oldest timestamp in block B's set. TSM's must use *true* least-recent-used (LRU) eviction, but may use any associativity.

Third, coherence protocol response messages (e.g., data and invalidation acknowledgements) include a *piggybacked timestamp*: a message payload not interpreted by the coherence protocol. This timestamp may be logged at the receiving core to order conflicting accesses.

Fourth, not shown, each core includes logging hardware to record selected timestamps and core identifiers. A simple implementation writes the log to the L2 cache for physical (or virtual) memory reserved for each core.

Fifth, optional *TSO support*, discussed later, requires that each core includes simple, local logic to detect when a read might violate SC to enable the value read to also be logged.

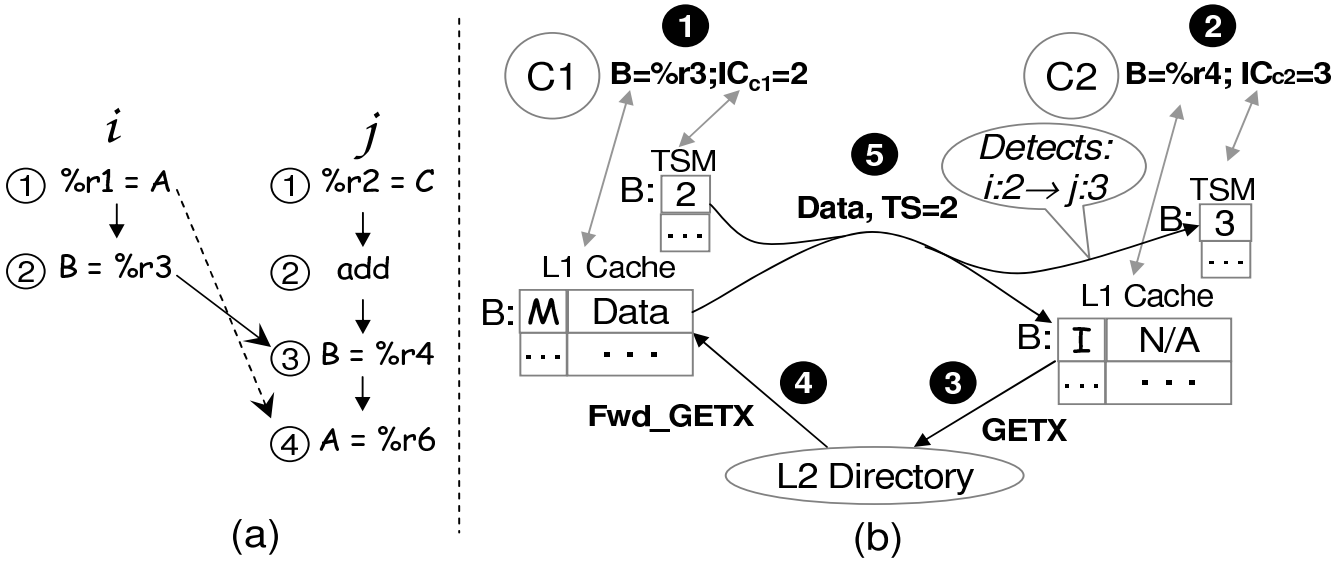

Figure 2: (a) a multithreaded execution and (b) FDR recording a shared memory race

Race Recording Operation. Consider the example provided in Figure 2 where part (a) depicts the code of two threads, *i* and *j,* and part (b) the sequence of FDR actions for block B only. Assume threads *i* and *j* run on cores C1 and C2, respectively. After thread *i* finishes its instruction 2 (*i*:2), block B is cached at core C1 as *modified (M)* and is *invalid (I)* at core C2.

1) As C1 writes memory block B, C1 records its current IC of 2 in its TSM as block B's timestamp.
2) When C2 seeks to write block B at IC 3, an L1 miss occurs.
3) C2 sends an exclusive coherence request (GETX) to the L2 directory.
4) The directory forwards the request to C1.
5) C1 looks up block B's timestamp of 2 from its TSM and responds to C2 with the data block B and a piggybacked timestamp.
6) Finally (not shown) C2 writes a "*i*:2 is before *j*:3" record into *j*'s log and completes its instruction 3, writing block B.

Later, a replayer can use this "*i*:2 is before *j*:3" record in *j*'s log to replay the race for block B in the same order. A replay (simulation) 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 stalling at thread *j*'s next log entry.

As discussed later, FDR will *not* need to record the dashed arc for block A from *i:1* to *j:4*. 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), *j:3* is before *j:4* (*j*'s program order), and the relation "is before" is *transitive* ("x is before y" and "y is before z" implies "x is before z").

Similar reasoning allows FDR's TSMs to be caches that don't explicitly store timestamps for all memory blocks. For example, let thread *i* access block B at IC 1000 and execute for sufficiently long that *i*'s TSM evicts B. Much later, if thread *j* accesses block B at IC_j, then *j*'s may log "*i*:2000 is before *j*:IC_j", where 2000 is provided by *i*'s TSM. This log entry is sufficient, because *i*:1000 is before *i*:2000 (*i*'s program order) and *i*:2000 is before *j*:IC_j (logged) imply *i*:1000 is before *j*:IC_j by transitivity.

FDR also assumes that is not acceptable to store timestamps in the shared L2 cache or to change the DRAM memory. It recovers missing timestamps with modest coherence protocol changes. After an L1 writeback of block B by C2, for example, the L2 directory entry for block B continues to remember C2. On a subsequent request for block by C1, the coherence protocol is modified to ask C2 to query its TSM for block B's timestamp (or larger approximation).

Why FDR's Race Recording Works

Here we sketch two alternative arguments why FDR's memory race recording is correct, but interested readers can find more detailed arguments in our prior work.

The first argument is that FDR's memory race recording works, because it records "the order" of coherence operations to enable the replayer to replay them in the same order. The same coherence order yields the same execution.

This argument was literally true for the recorder of Bacon and Goldstein [1]. They assumed a snooping system and recorded the total order of bus activity. FDR, on the other hand, only records a partial order of coherence activity. The order of conflicting accesses is recorded, but non-conflicting accesses may not be. While it seems intuitive that deterministic replay can be enabled without ordering non-conflicting accesses, proving it requires more formalism.

The second, more formal argument, due to Netzer [6], uses sequential consistency (SC) and a notion of equivalent SC executions. SC semantics require that the memory accesses

of all cores appear to be interleaved into a single total order. Two SC executions are *equivalent* if every read obtains the same value and the final memory state is the same. Netzer showed that recording the conflicting accesses of an SC execution, as FDR does, is sufficient to enable a replay of an equivalent SC execution.

How FDR Reduces Log Size Growth

FDR would not be viable if it created log entries on every interaction among cores, because doing so would require considerable write bandwidth and generate large logs. Log size can eventually limit how long race recording can operate. For fixed log size, dividing log growth rate by a factor R, multiplies recording length by the same factor R.

FDR actually optimizes log growth two ways. First, FDR simplifies Netzer's *transitive reduction (TR)* optimization [6] for hardware implementation [9]. We saw an example of TR in action when we discussed Figure 2. TR freed FDR from recording the dashed arc on block A, because it was implied by transitivity via program order and the recorded arc for block B.

Overall, TR is very effective, reducing log size growth by factors of 10 to 1000. This is because threads often run independently for many instructions and, when they do interact, often use a single synchronization variable to coordinate multiple data accesses. Let one thread access data blocks D1, D2, and D3 and then release lock L, while another thread waits for L to be freed, acquires it, and then accesses D1, D2, and D3. With TR, FDR will only log an entry for the block containing L, even though FDR is ignorant of program semantics (e.g., where locks are or what they mean).

Hardware for implementing TR is local and simple. Each core C_j records a table of the largest timestamp it received from each other core C_i. Core C_j does not log an entry from core C_i with IC_i if the timestamp it has stored for core C_i is greater or equal to the newly-received IC_i.

FDR improves on Netzer's original TR by (a) using TSMs to approximate missing timestamps, (b) tracking scalar, not vector, timestamps, and (c) providing a hardware implementation.

Second, FDR also uses an optional *regulated transitive reduction (RTR)* optimization to further reduce log size (e.g., 30%) [10, 8]. RTR often records stricter dependences than necessary for replay to (a) allow more log entries to be removed by TR and (b) compact entries with a process somewhat analogous to vectorization. Interested readers should consult the original work.

An Extension to Support SC and TSO

Previous race recorders, implemented in hardware or software, have required SC, making them inapplicable to most deployed hardware. We now extend FDR to support both SC and SPARC's total store order (TSO). We focus on

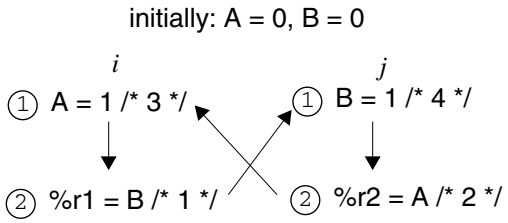

Figure 3: An example of TSO executions that are not SC, because writes are delayed by the write buffer (The numbers in /* */ denote the memory ordering.)

TSO, because it is well-defined and rumored to be implemented by many x86 systems as a valid implementation of processor consistency (PC). Our goal is to record TSO executions without forcing the executions to conform to SC, while, at most modestly increasing log size and hardware complexity. Readers uninterested in TSO may skip this section without loss of continuity.

TSO creates challenges for race recording. TSO relaxes write-to-read ordering to the shared memory. Informally, with TSO, a core can implement a hardware first-in-first-out (FIFO) *write buffer*. Figure 3 shows an example TSO execution that is not allowed by SC. In this execution, thread *i* first writes memory location `A` then reads a different memory location `B`, thread *j* first writes `B` then reads `A`. Because of the write buffers, the two reads are ordered before the writes are ordered. The numbers in *"/* */"* denote the memory ordering. For this execution, a race recorder that assumes SC 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 SC order, the recorded dependence cycle causes it to deadlock (cycle of dependencies).

We propose an *order-value-hybrid recorder* to handle SC and TSO executions [8]. Our key observation is that some reads cause replay deadlocks, because they are ordered (at memory) before writes that are earlier in program order. FDR's order-value-hybrid race recording detects and reacts to such *problematic* reads.

- It *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, ..., W_{n-1} (IC's < IC_n) are still in the write buffer and the cache block containing V is invalidated before all of the writes W_0, ..., W_{n-1} exit the write buffer.
- It *reacts* to a problematic read R by logging its instruction count IC_n and value read V.

This information later allows a replayer to, in effect, replay problematic reads *by value*, rather than *by ordering*.

Our problematic read detect logic is similar to that used in aggressive implementations of SC [2], but our reaction logs a tuple (and does not trigger a mis-speculation recovery to obtain SC compliance). Nevertheless, this logging (at most)

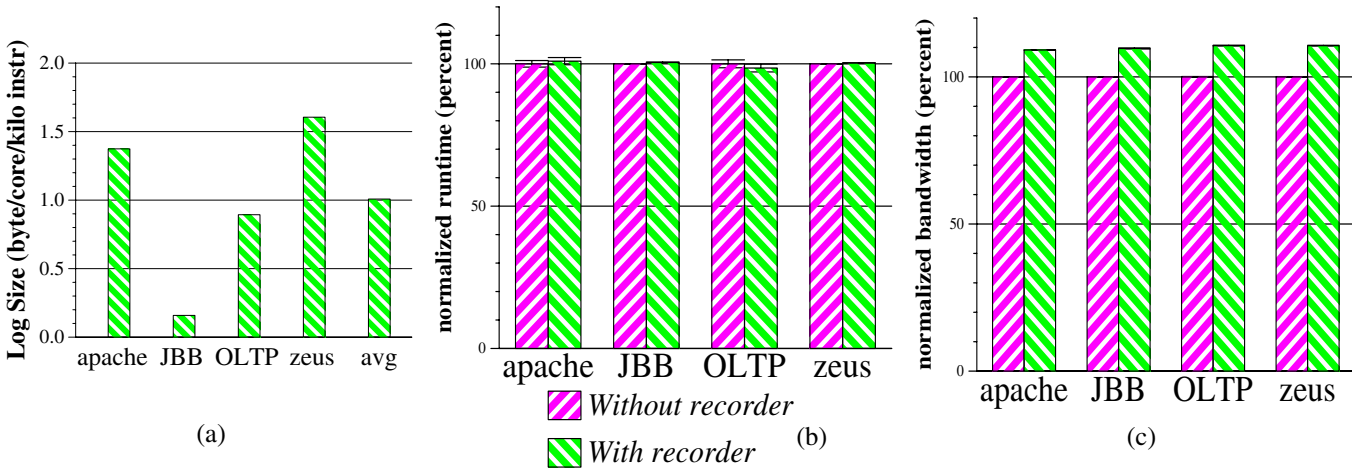

Figure 4: FDR's (a) log size growth rate, (b) runtime overhead, and (c) interconnect bandwidth overhead

slightly increases the log size, because reads that can potentially violate SC are known to occur infrequently.

FDR Memory Race Recording Methods & Results

Evaluation Methods. Here we summarize the methods we use to evaluate FDR's memory race recording. We use full-system simulation via Wisconsin GEMS [7] with Simics.

We model 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 16MB L2 cache, and 80ns off-chip DRAM. Each core is 1 GHz two-way-issue in-order with single-cycle split I&D 64KB L1 caches. Each core's TSM is 24KB to hold 2048 timestamps.

We exercise the system with four commercial workloads. *Apache* is a static web serving workload. *Online Transaction Processing (OLTP)* models database activities of a wholesale supplier, with many concurrent users performing transactions. *JBB* is a server-side java benchmark that models a 3-tier system, focusing on the middle-ware server business logic. *Zeus* is another static web serving workload.

Results. Figure 4 displays three key results. Part (a) shows that FDR's memory race log grows *about one byte per thousand instructions executed*. This allows FDR to record memory races for billions of instructions with megabytes of log storage. Part (b) reveals that FDR's memory race recording has negligible runtime overhead (less than 2%), while part (c) shows that it adds tolerable interconnect bandwidth overhead (about 10%).

In our judgement, these results support that FDR offers a practical option for recording races in future multicore systems.

Future Extensions

Our description of FDR race recording made many base system assumptions, such as single multicore system, single-threaded cores, directory coherence, writeback caches, and SC or TSO memory consistency. These assumptions can be relaxed in future work with varying degrees of effort.

Some extensions appear straightforward. Supporting systems with more than one multicore chip is trivial, depending the coherence protocols. Handling multithreaded cores (*e.g.*, hyper-threading) can be done by augmenting now-shared L1 caches with state bits to trigger pseudo-coherence logging events when different threads of the same core interact (but better solutions may exist).

FDR currently recovers timestamps of evicted blocks with modest coherence protocol changes. Alternatively, it might be better to leave the protocol unchanged by storing at an L2 cache bank the timestamps for when each core last wrote-back a block.

Other extensions require more creativity. Race recorders for systems with snooping, rather than directory coherence, may require other methods of reducing race logs, such as storing selected written-back timestamps, as above, or dividing memory accesses into temporal strata [4]. Moreover, while handling write-through caches to a private writeback cache appears easy, recording a system with write-through to a single shared cache may require more coherence protocol changes.

Finally, there is at least one extension without a known efficient solution: race recording with memory consistency models more relaxed than TSO. Xu [8] provides insight why our order-value hybrid does not directly apply.

Conclusions

Deterministic replay of multithreaded execution enables cyclic debugging on multicores. A key challenge is recording the original execution memory races without slowing down the execution significantly. To a multicore supporting SC or TSO, we propose adding modest "flight data recorder" (FDR) hardware that piggybacks timestamps on coherence messages to write optimized record on memory races using about one byte per thousand instructions executed.

While FDR requires hardware changes, perhaps the time has come to spend modest chip resources to ease debugging of the multithreaded software that we will all depend upon.

We thank Daniel Gibson and Michael Swift, as well as the people and groups acknowledged in our original work [9, 10, 8]. This work is supported in part by NSF with grants CCF-0085949, CCR-0093275, CCR-0105721, EIA/CNS-0103670, CCR-0105721, EIA/CNS-0205286, CNS-0225610, CCR-0243657, CCR-0324878, CCR-0326577; DARPA under contract No. NBCHC020056, awards from UC MICRO and Okawa Foundation, and donations from IBM, Intel, Microsoft, and Sun. Xu performed this work as a Wisconsin Ph.D. student, prior to joining VMware. Hill has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of DARPA, IBM, Intel, Microsoft, NSF, Sun, or VMware.

References

[1] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 194–206, 1991.

[2] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, Aug. 1991.

[3] T. J. Leblanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[4] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, Oct. 2006.

[5] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, June 2005.

[6] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 1–11, 1993.

[7] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[8] M. Xu. *Race Recording for Multithreaded Deterministic Replay Using Multiprocessor Hardware*. PhD thesis, University of Wisconsin, 2006.

[9] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–133, June 2003.

[10] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, Oct. 2006.

Min Xu is a Member of Technical Staff of VMware Inc. He is interested in hardware and software techniques in deterministic replay. He earned a PhD from University of Wisconsin-Madison. He is a member of the ACM and IEEE.

Rastislav Bodik is Assistant Professor in the Computer Sciences Division at the University of California, Berkeley. He was previously a member of faculty at the University of Wisconsin-Madison. His research interests include static and dynamic program analysis, software tools, and compilation. He earned a PhD from University of Pittsburgh.

Mark D. Hill is Professor in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin-Madison. His research interests include parallel computer system design, memory system design, and computer simulation. He earned a PhD from University of California, Berkeley and is a Fellow of the IEEE and an ACM Fellow.

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