A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay

Min Xu[1], Rastislav Bodik[1,2,3], Mark D. Hill[1,4]

[1]*Computer Sciences Dept. & ECE Dept. Univ. of Wisconsin-Madison mxu@cae.wisc.edu markhill@cs.wisc.edu*

[2]*Computer Science Div. EECS Dept. Univ. of California, Berkeley bodik@cs.berkeley.edu*

[3]*IBM T.J. Watson Research Center Hawthorne, NY 10532*

[4]*Computer Architecture Dept. Universitat Politècnica de Catalunya (UPC) Barcelona, Spain*

Abstract

Debuggers have been proven indispensable in improving software reliability. Unfortunately, on most real-life software, debuggers fail to deliver their most essential feature — a faithful replay of the execution. The reason is non-determinism caused by multithreading and non-repeatable inputs. A common solution to faithful replay has been to record the non-deterministic execution. Existing recorders, however, either work only for data-race-free programs or have prohibitive overhead.

As a step towards powerful debugging, we develop a practical low-overhead hardware recorder for cache-coherent multiprocessors, called ***Flight Data Recorder (FDR)****. Like an aircraft flight data recorder, FDR continuously records the execution, even on deployed systems, logging the execution for post-mortem analysis.*

FDR is practical because it piggybacks on the cache coherence hardware and logs nearly the minimal thread-ordering information necessary to faithfully replay the multiprocessor execution. Our studies, based on simulating a four-processor server with commercial workloads, show that when allocated less than 7% of system's physical memory, our FDR design can capture the last one second of the execution at modest (less than 2%) slowdown.

1. Introduction

An important challenge facing computer industry is to develop reliable software and to maintain it after deployment. Debugging has been an effective solution to this challenge. Essential to any debugger, *deterministic replay* enables an expert to re-execute the program and zero in on bugs that faithfully re-appear. One problem in debugging that has been eluding a software-only solution is how to perform deterministic replay of multiprocessor executions. As hardware costs diminish relative to software and maintenance costs [17], it is economical to seek hardware that facilitates debugging of complex systems.

A common method for deterministic replay of multiprocessor executions is to record outcomes of all non-deterministic events, most notably those of memory races. The key problem addressed in this paper is how to record memory races with small overhead, to enable recording on deployed systems. Low-overhead recording has been an open problem (see Section 2): Several systems record *synchronization* races, but without recording *data* races, they can faithfully replay only data-race-free programs. Unfortunately, programs being debugged may not identify synchronization or may contain data races, harmful or harmless, and these races must be recorded for deterministic replay.

Furthermore, no existing system supports deterministic replay in the harsh *full-system* environment, where applications and diverse I/O devices from different vendors interact via operating system mechanisms on multithreaded hardware. To illustrate the difficulties, consider finding a bug that manifest itself only when a device by a vendor "A" interrupts a driver by vendor "B" between two instructions of what should have been an atomic update!

To enable full-system deterministic replay of multiprocessor executions, we propose a **Flight Data Recorder (FDR)** that adds modest hardware to a directory-based sequentially consistent multiprocessor. Like an aircraft flight data recorder, FDR continuously records the execution in anticipation of a "trigger", which can be a fatal crash or an software assertion violation. When triggered, FDR produces a *log-enhanced core dump* for a *replay*

This work is supported in part by the National Science Foundation, with grants EIA-9971256, CCR-0093275, EIA-0103670, CCR-0105721, and EIA-0205286, a Wisconsin Romnes Fellowship (Hill), Universitat Politècnica de Catalunya and Secretaría de Estado de Educación y Universidades de España (Hill sabbatical), and donations from IBM, Intel, Microsoft, and Sun.

interval preceding the trigger, to be analyzed in an off-line replayer (not developed in the paper). The core dump includes three kinds of logs, each designed to meet performance, space, and complexity requirements of a practical recorder for continuous recording in deployed systems:

1) To restore a *consistent system state* at the beginning of the replay interval, FDR modestly adapts an existing approach that logs old memory state whenever the memory is updated [26], and dumps the complete memory image at the time of the trigger (Section 3).

2) To record the *outcomes of all races*, FDR logs a (nearly minimal) subset of the races (Section 4). We build on hardware ideas of Bacon and Goldstein [2] and the software transitive-reduction optimization of Netzer [15]. This optimization avoids logging races whose outcomes are implied by other races. A limitation of our design is its assumption of sequential consistency (Section 4.3).

3) To record *system I/O*, FDR logs interrupt timing and treats device interfaces as pseudo-processors to ease DMA logging (Section 5).

We propose a concrete FDR design, called FDR1, (Section 6) and evaluate its practicality using full-system simulation of a four-processor system with four commercial workloads (Section 7). We qualitatively discuss scaling to other systems and workloads (Section 8).

Our experiments show that low time and space overhead allows FDR1 to be continuously enabled. Specifically, we find that for a system with modern processors and 1GHz system clock, FDR1 generates logs at the bandwidth of 25 MB/second/processor. (At the time of the trigger, the logs are copied from memory, together with the memory image, to an external device, e.g., a disk.) The net result is that FDR1 can capture a replay interval of one billion cycles prior to the trigger with only 7% of the system physical memory dedicated to its logs.

The paper concludes with a discussion of how an offline software *replayer* would use FDR's logs to perform deterministic replay (Section 9).

This paper makes two key contributions:

- *It designs an efficient full-system recorder, called FDR, for sequentially consistent multiprocessors.* FDR allows deterministically replaying full-system activities (including I/O) of a system with speculative processors. FDR is easy to implement because it piggybacks on the directory-based cache-coherence protocol. FDR is efficient because it records a small (nearly minimal) subset of all races in the system.

TABLE 1. Summary of key related work.

Name	**Enables Deterministic Replay**	**Handles OS & I/O**	**Hardware Support**
Goal: Deterministic Replay			
Flight Data Recorder	Always[1]	Yes	Yes
InstantReplay [10]	Yes	No	No
Bacon & Goldstein [2]	Yes	No	Yes
Netzer [15]	Yes	No	No
Deja Vu [4]	Uni-Proc	No	No
TraceBack [6]	Partially[2]	No	No
Goal: Race Detection			
Dinning&Schonberg [5]	No	No	No
Min & Choi [14]	No	OS	Yes
Eraser [22]	No	No	No
Richard & Larus [20]	No	OS	No
RecPlay [21]	Sometimes[3]	No	No

1 Within designed replay interval.

2 Only basic block execution history.

3 If synchronizations are identified & no data races.

- *It evaluates the recorder on commercial workloads via full-system simulations.* The simulations show that FDR is practical: it generates relatively small logs and its modest time overhead (less than 2%) allows it to be continuously enabled.

2. Related Work

This section relates FDR to existing work, which we divide into deterministic replayers and data-race detectors. The former are related to FDR because they all contain a recorder of non-determinism; the latter are related because replaying detected races is sufficient to faithfully replay a multithreaded execution.

The upper half of Table 1 lists techniques for deterministic replay. In most significant contrast to FDR, these techniques do not handle operating system and I/O issues, which makes them unsuitable for full-system debugging of OS code in the presence of non-repeatable system input. The software-based deterministic replay technique InstantReplay [10] records the memory access order by means of variable versioning, generating large logs, especially for programs with fine-grain synchronization. The recording techniques by Bacon and Goldstein [2] (the only hardware recorder we know of) reduce overhead of software methods by exploiting snooping-based coherence, but they log information on every coherence transaction. FDR improves on their work by developing a variation of Netzer's [15] transitive reduction to suppress logging most coherence transactions (see Section 4.1). Furthermore, FDR operates in a distributed fashion, without a centralized bus or log buffer. Further reducing the logged information is difficult without access to high-level program semantics. DejaVu [4], for example, assumes multi-threaded programs running on a single processor system, which allows it to limit its recording to scheduler decisions. Lastly, TraceBack [6] is a commercial software product that, from the limited documentation available to us, appears to (faithfully) record the control-flow path taken by each thread, but not the memory race interleaving among threads.

The bottom half of Table 1 lists detectors of data races, as defined in Netzer and Miller [16]. Dinning and Schonberg [5] developed an online monitoring algorithm for detecting data races in Fortran doall/endall constructs, based on an efficient execution-history compression technique. Their compression later became the foundation of Netzer's transitivity reduction [15]. Even lower overhead was achieved with a hardware race detector of Min and Choi [14]. Based on cache coherence, this detector timestamps cache blocks with synchronization "era" (with respect to fork/join/barrier synchronizations). The idea of Richards and Larus [20] is similar to that of Min and Choi, except that they build on a more flexible software memory coherence protocol. Later, the Eraser detector [22] proposed to detect improperly locked shared variables (not necessarily races) by keeping track of the set of locks that protected shared variables during the execution. Finally, RecPlay [21] records synchronization races and finds the first data race in the execution using a more expensive off-line analysis. RecPlay can also perform deterministic replay, but only up to the first race; other methods in this group do not record enough information for replay.

3. Recording System State

The first of FDR's three logs ensures that an offline replayer can re-construct the *initial replay state*, i.e., the system state at the beginning of the replay interval. The initial replay state includes the architectural state of all processors (e.g., registers, TLBs), the physical memory image, and, optionally, the I/O state (Section 5).

A key challenge is that FDR does not know when a replay interval starts, because it cannot predict when a trigger (e.g., crash) will occur. Fortunately, this problem has been solved many times before for backward error recovery [7, 18]. Conceptually, a recorder periodically saves a complete copy of the system state, called a *checkpoint*. An old checkpoint can be discarded as soon as there is a new checkpoint old enough to begin a sufficiently long replay interval.

FDR uses a common method of incrementally creating logical checkpoints, called *logging*, which saves the old value of memory locations before they are updated. When needed, a logical checkpoint is recovered by taking the final system state and restoring the old values in reverse log order. A common optimization is to only log the first update for each memory location in each logical checkpoint.

Ideally, FDR's checkpoint scheme should 1) operate with acceptable overhead since the recorder is "always on"; and 2) operate with the cache-coherent shared-memory multiprocessors. Fortunately, two recently developed approaches, ReVive [19] and SafetyNet [26], meet FDR's checkpoint needs. In this paper we evaluate FDR using a variation of SafetyNet, but we leave finding the best checkpoint solution to future work.

4. Recording Memory Races

Starting from the initial replay state (i.e., the checkpoint), FDR must ensure that each replayed instruction produces the same values as in the original execution. For single-processor systems, it is sufficient to log system inputs (Section 5). For multiprocessor systems, it is also necessary to log non-deterministic thread interleaving — i.e., the outcomes of races.

The key question underlying the FDR design is how much of the execution must be logged. The answer depends on the memory model. FDR developed in this paper assumes a *sequentially consistent* (SC) memory system. In an SC execution, all instructions from all threads (appear to) form a total order that is consistent with the *program order* on each thread. In this total order, a load get value last stored to an address [9].[1]

To replay an SC execution, it is trivially sufficient to log the total instruction order observed in the execution, by recording *arcs* that order all pairs of dynamic instructions. Many such arcs are, however, not necessary for replay. First, arcs between instructions that access different memory locations can be omitted because replaying independent instructions out of order preserves computed results. Second, many arcs can be omitted because they are implied by other, more constraining arcs.

1. This definition applies even to systems that support unaligned, variable-sized memory operations via modeling each B-byte load (store) to address A as an atomic sequence of byte loads (stores) to addresses A, A+1,..., and A+B-1.

These observations are formalized in Netzer's software recorder [15], which records a provably minimal set of ordering arcs. Our design is a hardware variation on Netzer's recorder. Most notably, in order to deliver a simple, low-overhead recorder, we decided to record a larger set of arcs. In this section, we first illustrate the basic idea with an example and develop FDR for idealized hardware. We then extend FDR to realistic hardware.

4.1. An Example

We begin by examining the information that must be logged to replay the SC execution of Figure 1. The figure illustrates two processors, *i* and *j*, each executing instructions labeled by a local dynamic instruction count. We say that two accesses have a *word (block) conflict* if they access the same word (block) and at least one is a store.

In an SC execution, all instructions appear to execute in a total order. Figure 1(a) shows arcs that specify the total order listed in the caption. Note that the total order includes the intra-processor program-order arcs.

Let us now show that not all arcs in this total order need to be recorded to faithfully replay the execution. First, deterministic replay can be ensured without recording any program order arcs, because they can be trivially reconstructed during replay.

Next, it is not necessary to record the order of independent instructions, e.g., *j*:16→*i*:33, because no thread execution will change if the order is reversed. Figure 1(b) removes arcs of such order, leaving only word conflict arcs. These arcs are sufficient, but again not necessary.

Figure 1(c) further improves by recording block conflict arcs. For example, when variables "Share1" and "Share2" are in the same cache block, word conflict arcs *i*:32→*j*:21 and *i*:33→*j*:22 are replaced by one block conflict arc *i*:33→*j*:21. Nevertheless, *i*:33→*j*:21 and program order arcs *i*:32→*i*:33 and *j*:21→*j*:22 still imply *i*:32→*j*:21 and *i*:33→*j*:22 by transitivity (*i*:32→*i*:33→ *j*:21→*j*:22). Also note the new arc (*j*:14→*i*:35) created when variables "Local1" and Local2" reside in the same cache block.

Finally, Figure 1(d) removes block conflict arcs that are implied by transitivity of other block conflict arcs and program-order arcs (e.g., arc *i*:33→*j*:21 is removed because it is implied by *i*:33→*i*:34→*j*:18→*j*:19→*j*:20→ *j*:21). Note that the word conflict order is still preserved. Thus, this example can be deterministically replayed by recording only three arcs!

The optimizations shown in this example are valid for all SC executions, as shown in [5, 15]. In summary, there are three results:

- **From SC to Word Conflict.** Deterministic replay can be enabled by recording program order on each processor and word conflict order between processors.

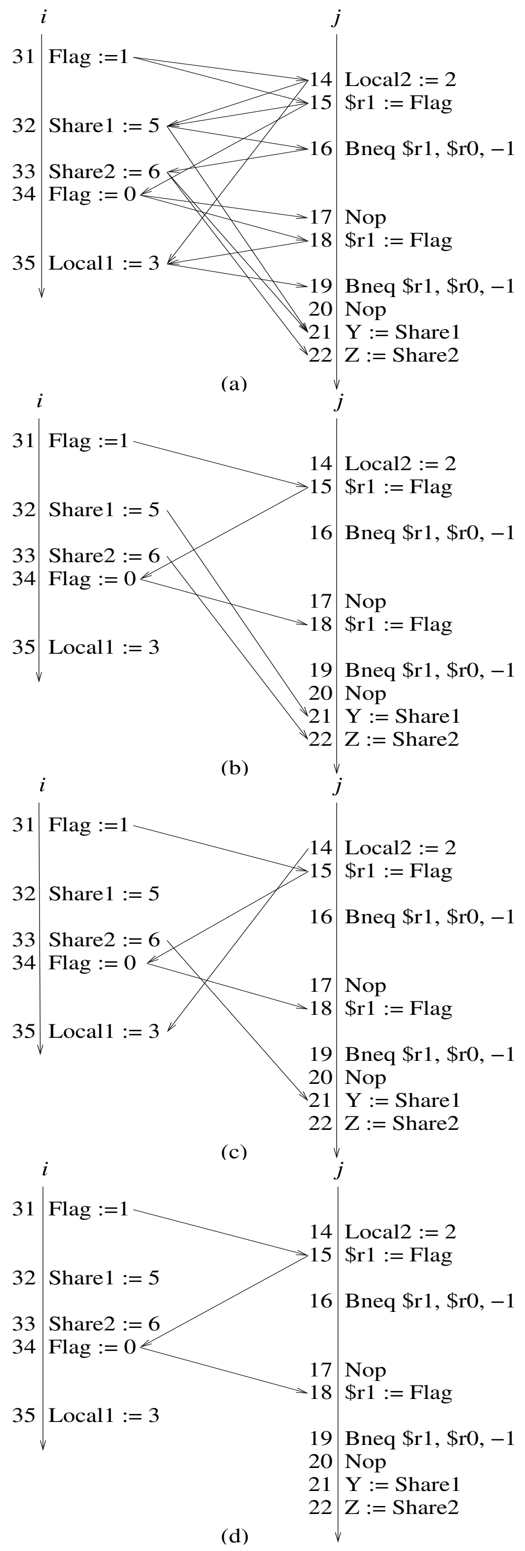

FIGURE 1. Four Views of an SC execution with total order: *i*:31 → *j*:14 → *j*:15 → *i*:32 → *j*:16 → *i*:33 → *i*:34 → *j*:17 → *j*:18 → *i*:35 → *j*:19 → *j*:20 → *j*:22.

- **From Word Conflict to Block Conflict.** It is sufficient to record program order and block conflict order.
- **From Block Conflict to Transitive Reduction.** It is sufficient to remember the transitive reduction of program-order and block conflict order. The *transitive reduction* of {a<b, b<c, a<c} is {a<b, b<c}, because a<c is implied by transitivity.

4.2. Idealized Hardware

This section and Section 4.3 translates Netzer's work into hardware for our base multiprocessor. The translation enables hardware to record block conflicting arcs by using its cache and cache coherence messages. To make the translation more straightforward, we initially simplify the problem in two ways. First, let each processor's cache be as large as memory (no cache block replacements) and all integers be unbounded (no wraparound). These assumptions ensure that the hardware does not forget information about past conflicts. Second, assume that each processor presents memory operations to its cache in program order, the cache completely handles one operation before starting the next one (blocking cache), and neither the processor nor the cache do any prefetching. These assumptions ensure that if a cache has a coherence request outstanding, the request is by the next instruction its processor wishes to commit. We remove these assumptions in Section 4.3.

We develop our techniques for directory protocols where processors always act on incoming coherence messages before send outgoing messages (which precludes some subtle buffering of invalidations [23]). We exploit two properties of a directory protocol. First, *coherence messages reveal the arcs in the SC order*. Let processor *i* execute some instructions i_1 to i_n, then send a coherence message to processor *j*, the interconnect delivers the message to *j*, and *j* processes the message before executing instructions j_1 to j_m. In this case, instructions i_1 to i_n will appear in SC execution order before j_1 to j_m. Second, *a directory protocol will reveal all block conflict arcs*, because if two accesses from different processors conflict at the block granularity, the accesses are always ordered by intervening coherence messages. Otherwise, coherence will be violated with one processor storing a block that another processor could still be loading or storing. For example, let's assume processor *j* requests an modified copy (state M) of a block that is currently non-exclusively owned (O) by processor *i* and read-only shared (S) by processors *k* and *l*. Order is provided by processor *i*'s data response to *j*, processor *k*'s invalidation acknowledgment to *j*, and processor *l*'s invalidation acknowledgment to *j*.

Table 2 illustrates the per-processor state and actions sufficient to log memory order in our idealized system. We augment each processor *j* with an *instruction count*, **IC**; a *log buffer*, **LOG**; and, a *vector of instruction counts*, **VIC[P]**, with an entry for each other processor. We also augment each cache block frame b at processor *j* with *cache instruction count*, **CIC[b]**. The actions illustrated in Table 2 piggyback information to each coherence reply to enable the receiver to log a block conflict arc, if it is not implied by transitivity. Specifically, each processor *j* takes the following actions:

- On instruction commit, it updates its IC and, if the instruction is a load or store, the corresponding CIC.
- When sending a coherence reply, *j* includes its own identifier and the IC when it last accessed the block being transferred (*j*:CIC[b]). Thus the coherence reply identifies the *beginning* of a block conflict arc.

TABLE 2. State and Actions for Processor *j* to Log Memory Order.

State at each processor *j*

IC: Instruction count of last dynamic instruction committed at processor *j*.

LOG: Unbounded log buffer at each processor *j*.

VIC[P]: Vector IC for transitive reduction: VIC[*i*] retains latest IC received by *j* from *i*. **P** is the total number of processors.

CIC[M]: Cache IC: CIC[b] gives IC of last load or store of block b in *j*'s cache. **M** is the total number of memory blocks.

Actions at each processor *j*

```
On commit of instruction insn {
    IC++ // After, IC is insn's dynamic instruction count
    if (is_load_store(insn)) {
        // b must be cached before insn can commit
        b = block_accessed(insn)
        // CIC[b] = the last IC to access block b
        CIC[b] = IC
    }
}
On sending coherence reply for block b to proc k {
    // Arc begins at processor j's last instruction to access
    // block b, which is j:CIC[b]
    send.id = j
    send.ic = CIC[b]
    send(send.id, send.ic, ...)
}
On receiving coherence reply for block b from proc i {
    // Arc ends at the next instruction processor j will
    // commit, which is j:(IC+1)
    receive(rec.id, rec.ic, ...)
    if (rec.ic > VIC[i]) {
        // Transitive reduction: only log arc if it began
        // at i later than last arc received from i.
        Append to LOG =
            (rec.id, rec.ic, IC+1)
        VIC[i] = rec.ic
    }
}
```

- When receiving a coherence reply from *i* (e.g., data response or invalidation acknowledgment), *j* recognizes that the block conflict arc will *end* at the next instruction it wishes to commit (*j*:IC+1). It logs the arc if it is *not* implied by transitivity.

The transitive reduction works as follows. The new arc began at *i* at a CIC recorded in message field rec.ic. VIC[*i*] records the largest rec.ic that has previously been sent from *i* to *j*. If rec.ic is greater than VIC[*i*], then the arc is logged and VIC[*i*] is updated. Otherwise the new arc is implied by transitivity, since there must be an old conflict arc whose rec.ic was larger (and is already recorded in VIC[*i*]), but was already processed at *j* (with a smaller IC than for the new arc's termination).

Our hardware implementation of the transitive reduction is a variation of Netzer's optimal algorithm. To implement his original algorithm, both the cache (i.e., the CIC array) and the coherence messages would have to contain the entire VIC vector, as opposed to only the local IC value. With this simplification, we lose some information about which arcs are implied by already logged arcs. This modestly reduces the effectiveness of the transitive reduction on the workloads we have studied. This simplification, however, greatly reduces FDR's hardware overhead.

In summary, our algorithm allows the idealized system of this section to use coherence hardware to implement the techniques Netzer proved correct. Next, we seek to make the hardware practical.

4.3. Realistic Hardware

This section extends the algorithm for idealized hardware to handle the "implementation details" of speculative processors, finite non-blocking caches, and finite integer fields of a realistic system. Note that the realistic system still implements the SC memory model. We use send and receive observations as lemmas for optimizations.

Send Observation. *When a processor j at instruction count IC sends a coherence reply for block b, it may include any instruction count in the interval [CIC[b], IC].*

In Section 4.2, processor *j* always included CIC[b], the instruction count when *j* last accessed b. This action ordered all instructions that processor *j* executed up to instruction CIC[b] before the block conflict arc of this message. In fact, however, processor *j* has already executed instructions up to its current instruction count, IC (IC ≥ CIC[b]). So *j*:IC is actually ordered before the message in the SC execution. Thus, *j* can alternatively send any value in the interval [CIC[b], IC]. Most of these values will not begin a block conflict arc (because their instructions don't access block b), but they are arcs in the SC execution and always imply the block conflict arc that begins at CIC[b] by transitivity. We apply the send observation by always sending the current IC. However, this modestly reduces the effectiveness of the transitive reduction. In the extreme, eliminating all CIC's is permitted, but doing so would render the transitive reduction ineffective.

Receive Observation. *When a processor j at instruction count IC receives a coherence reply for block b, it may associate the arc with the next instruction it wishes to commit, IC+1, regardless of whether the next instruction (IC+1) accesses block b or not.*

In Section 4.2, processor *j* associated the arc with instruction IC+1, but was assured that IC+1 accessed block b to terminate a block conflict arc. This assurance is not necessary. Instead, we can allow *j* to associate the arc with IC+1 regardless of whether it accesses b. This obtains an arc of the SC execution that is not necessarily a block conflict arc. If *j* subsequently accesses b at IC′ (IC′ > IC+1) the recorded arc will imply the block conflict arc terminating at IC′ by transitivity. If *j* never accesses b, then an unnecessary arc is recorded. Processor *j* cannot access b before IC+1, because *j* will stall for coherence permission.

It is important that even with send and receive observations, we do not accidentally record conflict arcs that form a cycle with program order arcs in the execution graph. Such a cycle can cause replayer deadlock. Fortunately, the ordering arcs of an SC execution cannot form a cycle [24]. Thus, the arcs recorded correspond to an acyclic graph, since all SC arcs form an acyclic graph, and its transitive reduction remains acyclic.

Speculative Processors. In a speculative processor, non-blocking caches and out-of-order execution allow coherence requests for blocks that are *not* accessed by the next instruction to commit. In fact, a cache block may be speculatively accessed before its coherence message arrives. A processor implementing SC, however, appears to access blocks when instructions commit; therefore, an early access will always appear to follow the message. As a result, the receive observation continues to hold (i.e., if we associate each coherence reply with IC+1, the actions in Table 2 continue to operate correctly).

Finite Caches. Table 2 assumes each processor's cache could cache all M memory blocks, whereas real caches can only hold C blocks where C << M. The latter must replace blocks and will have cache instruction counts only for currently cached blocks. Assume each processor *j* can silently replace shared blocks. When a later request from *k* seeks to invalidate a block *b* that *j* no longer caches, *j* sends an acknowledgment with its current instruction count, IC (since *j* no longer has a CIC entry for the block). The correctness is shown by the send observation. Similarly, each processor *j* can write back owned blocks to the directory at memory. The directory leaves *j* in its now-unused owner field to denote the last writer. Until the block is owned again, subsequent coherence requests ask *j* for its current IC. We find that this situation occurs rarely, since actively shared blocks are rarely written back.

Unordered Interconnect. Table 2 assumes at most one message is been transmitted from processor *i* to *j* in any given time. With speculative processors, multiple messages from processor *i* can arrive at processor *j* out of order. However, since each message contains the entire IC (as oppose to a change of IC) for the beginning of the arc, we associate the ending of the arc with the current committing instruction. Receive observation ensures the correctness. Nevertheless, message re-ordering can affect the effectiveness of the transitive reduction.

Finite Integer Fields. Table 2 assumes unbounded counters, while practical hardware must use finite counters. A trivial solution is to use 64-bit counters since they will not overflow for decades. When a recorder uses checkpointing (to solve Section 3's problem), fields can be much smaller, since they can be reset at checkpoints. If checkpoints always occur less than four billion instructions apart, for example, 32-bit unsigned ICs are sufficient.

Moreover, the CIC field in each cache block can be allowed to wraparound to both avoid resetting them at checkpoints and allow them to be small (e.g., 16 bits). With a coherence reply, always send min((IC & 0xffff0000)|CIC[b], IC). Since the first term is at least as large as the true CIC, the value sent is in the interval [CIC[b], IC], as is allowed by the send observation.

Sequential Consistency. A significant limitation of the FDR designed in this paper is that it assumes sequential consistency (SC). The design will also work in systems where a relaxed memory consistency model is implemented with SC (as is allowed). However, the current design does not work for relaxed implementations that allow some memory operations to execute out of program order, in part because it assumes a single instruction count is sufficient to indicate which of a processor's memory operations have executed. A recorder for relaxed implementation is possible (e.g., by completely recording the order of memory operations and coherence transfers), but future work is needed to make such a recorder efficient.

5. Recording Input/Output

The third task of FDR is ensuring that the log-enhanced core dump contains sufficient information to enable an offline replayer to deterministically replay I/O events in the replay interval. This section discusses the information FDR saves, assuming the offline replayer cannot re-create the input content (e.g., from a remote source), but can model a processor taking an exception, provided it is given the cause (interrupt or trap).

Program I/O. Most systems allow processors to execute loads and stores to addresses mapped to devices (I/O space). FDR logs the values returned by I/O loads, so they can be replayed. It does not log I/O store values, because they can be regenerated during replay.

Interrupts and Traps. *Interrupts* are asynchronous events (e.g., disk read completion) that may arrive at a processor between any two instructions. FDR logs both the content (e.g., interrupt source) and instruction count of when an interrupt arrives. This gives the replayer sufficient information to invoke the same exception.

Traps are synchronous events caused by a specific instruction interacting with system state (e.g., a TLB miss). FDR logs neither the content nor the order of traps, because they can be inferred by a replayer that exactly models the system. Alternative designs could log some traps to simplify replayers.

Direct Memory Access (DMA). Most systems allow *DMA writes* (*DMA reads*) whereby devices write (read) a series of memory blocks without involving processors. For each block in a DMA write, we assume a DMA interface updates memory and has the directory protocol invalidate any previously cached copies. For a DMA read, the DMA interface obtains the latest version of the block.

Our FDR logs DMA by modeling DMA interfaces as pseudo-processors and applying the memory-race logging techniques of Section 4. When a DMA interface seeks to write a block, it logs the ICs from invalidation acknowledgments (to order the write after previous operations), logs the data written, (since it comes from outside the system), increments its IC (treating the write like a block-size store instruction), and has the directory remember it as the previous owner (to order the write before subsequent operations). A DMA read operates similarly, but doesn't log data, because it can be regenerated during replay.

6. FDR1: Example Recorder Hardware

This section provides a concrete example of the Flight Data Recorder system. Figure 2 illustrates the system, which we call FDR1. FDR1 adds to a directory-based SC multiprocessor several structures shown in shaded boxes. The design stores all logs in physical memory. Evaluation will show that FDR1 enables the design goal of being able to replay intervals of one second. Below we describe how the structures function to implement FDR1.

Recording System State. FDR1 creates logical checkpoints to begin potential replay intervals (Section 3) by adapting SafetyNet [26]. SafetyNet creates a logical checkpoint with a physical checkpoint of processor registers and by logging cache blocks in Cache and Memory Checkpoint Log Buffers. A checkpoint number field in

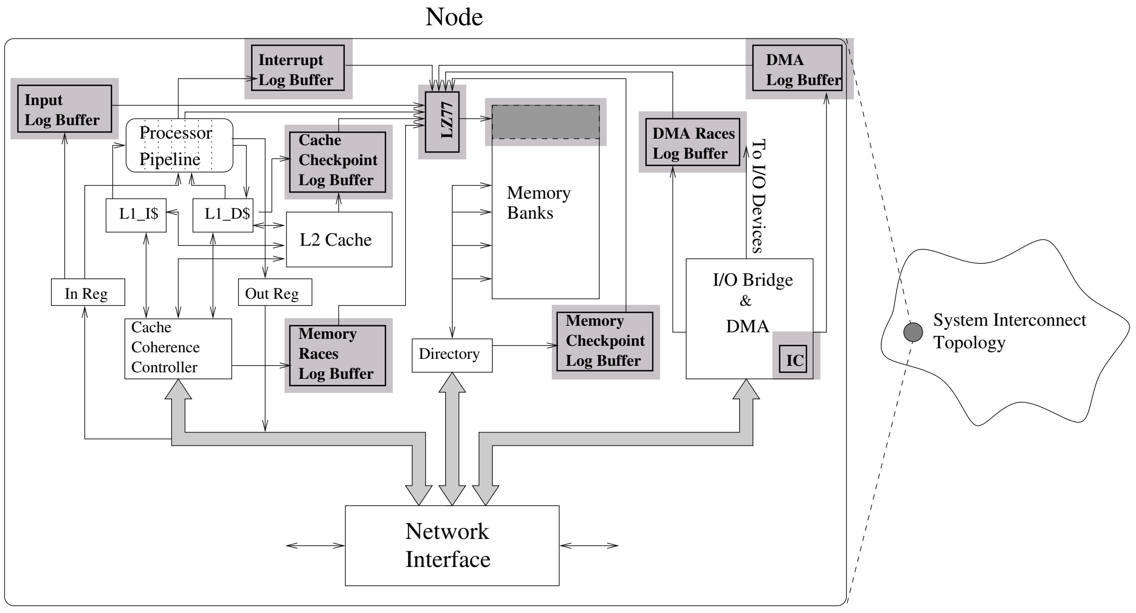

FIGURE 2. Example MP system with the Flight Data Recorder. We assume a base system that implements sequential consistency using per-processor cache hierarchies and a MOSI write-invalidate directory protocol. The system supports conventional I/O. For FDR1: The Input Log Buffer and Interrupt Log Buffer are close (on the same die) to processor core. Race Log Buffer and Cache Checkpoint Log Buffer are close to cache. Memory Checkpoint Log Buffer and DMA Buffers are close to memory controller and DMA interface, respectively.

each cache block ensures a block is only logged the first time it is updated or transferred after a checkpoint. FDR1 reduces the checkpoint number to a checkpoint bit, since it only recognizes one active checkpoint.

FDR1 requires logical checkpoints much further back (1 second) than was the focus of the original SafetyNet design (<1 ms). For this reason, as Figure 2 illustrates, FDR1 uses Cache and Memory Checkpoint Log Buffers only to tolerate bursts, then compresses the data using Lempel-Ziv [29] implemented in hardware (LZ77 box), and finally saves it in physical memory that is reserved from the operating system (shaded). Logging a memory/cache block takes 72 bytes for 64-byte data and 8-byte physical address. Other hardware implementations of the data compressor are also possible [27].

Since FDR1 checkpoint frequency is relatively low, the overhead to start one checkpoint can be tens of thousands of cycles without significant impact to normal execution. For this reason, FDR1 quiesces the system to take a checkpoint (less aggressive than SafetyNet) using a standard 2-phase commit (as in ReVive [19]). When a cycle count reaches a checkpoint threshold, a processor completes current activities, marks the end of the log for the previous checkpoint interval, and tells a system controller it is ready. When the system controller notifies that all processors are ready, a processor writes its register and TLB state to physical memory (4248 bytes for SPARC V9 with 64-entry I/D TLBs), clears a checkpoint bit in each cache block [13, 25]. Each processor then coordinates with the system controller to complete the checkpoint handshake.

For a replay interval of one second, a reasonable policy is to take a logical checkpoint every 1/3 second and reserve physical memory to store execution logs for four checkpoints. When the second-oldest checkpoint is 1 second old, the oldest checkpoint is discarded, so that its storage can be used for the logs of the next checkpoint. This policy provides a replay interval of between 1 and 4/3 seconds, depending on timing of the trigger.

Recording Memory Races. FDR1 records memory races using the algorithms in Section 4. As Figure 2 illustrates, it logs information on memory races in the Memory Race Log Buffer, which is then compressed with Lempel-Ziv and stored in reserved physical memory. Each entry is 9 bytes: 1 byte for the other processor's ID, 4 bytes for the other processor's IC, and 4 bytes for this processor's IC. As Figure 3 illustrates, the processor core is augmented with an IC field, the cache controller is augmented with VIC fields, and each cache block frame is augmented with a CIC entry. With 32-bit CIC fields and 64-byte cache blocks, the cache overhead is 6.25%. IC, CIC, VIC are reset at checkpoint boundary.

Recording I/O. FDR1 records I/O with the algorithms in Section 5. It logs input data from I/O loads in the Input Log Buffer (8-bytes per entry) and interrupt numbers in

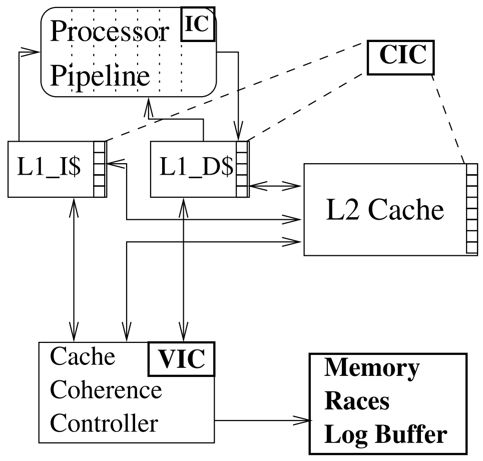

FIGURE 3. FDR1 processor/caches.

the Interrupt Log Buffer (4-byte IC and 2-byte interrupt number per entry). Additionally, DMA engines in the system act like pseudo-processors with their own IC and VIC. They log entries in the DMA Race Log Buffer (9-byte entry as in the Memory Race Log) to order DMA. And they log DMA store values in the DMA Log Buffer (8-byte address and variable-sized data).

The Trigger. When triggered (e.g., on a crash), the FDR1 system controller first stops processor execution and flushes all log buffers. Next, it copies the contents of all physical memory to an external device, such as a disk. Since all logs are stored in the memory, this core dump constitutes the log-enhanced core dump that enables deterministic replay. The size of the log-enhanced core dump is dominated by the size of the memory image core dump, since logs are a small fraction of all physical memory.

7. Evaluation Methods

We evaluate FDR1 for a four-processor machine using execution-driven multiprocessor simulation. We obtain the functional behavior of a multiprocessor Sun SPARC V9 platform architecture (used for Sun E6000s) using Virtutech Simics [11]. Simics mimics a SPARC system in sufficient detail to boot unmodified Solaris 9.

We obtain performance estimation by extending the Wisconsin performance models [1, 12]. To facilitate commercial workload simulation of adequate length to stress FDR1, we model a simple in-order 1-way-issue 4 GHz processors with 1 GHz system clock. Despite SPARC V9 architecture specifying *Total Store Order* (TSO) as the default memory model, our simulator implements *Sequential Consistency* as a correct implement of TSO. We model a MOSI directory protocol, similar to that of the SGI Origin, with and without FDR1. The simulator captures all state transitions (including transient states) of the coherence protocol in the cache and memory controllers. We

TABLE 3. Target System Parameters.

Number of Processors	4
L1 Cache (I and D)	128 KB, 4-way
L2 Cache	4 MB, 4-way
Memory	512MB/processor, 64Bblocks
Miss From Memory	180 ns (uncontended, 2-hop)
Interconnection	2D torus, link B/W=6.4 GB/s
Memory Reservation	34 MB (7% of 512 MB)
Memory Checkpoint Log Buffer	256KB, output B/W=70MB/s, 72B entries
Cache Checkpoint Log	1024 KB, 50 MB/s, 72B
Memory Race Log	32 KB, 10 MB/s, 9B
Interrupts Log Buffer	64 KB, 10 MB/s, 6B
Input Log Buffer	8 KB, 10 MB/s, 8B
DMA Log Buffer	32 KB, 10 MB/s, variable size
Checkpoint Interval	333,333,333 cycles = 1/3 sec.
System Barrier Cost	10,000 cycles = 10 us

model a 2D-torus interconnection as well as the contention within this interconnect, including contention with and without FDR1 message overhead.

We exercise FDR1 with four commercial applications, described briefly in Table 4 and in more detail by Alameldeen et al. [1]. We also adopt that paper's approach of simulating each design point multiple times with small, pseudo-random perturbations of memory latencies to miti-

TABLE 4. Wisconsin Commercial Workloads.

OLTP: Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 800 MB 4000-warehouse database stored on five raw disks and an additional dedicated database log disk. There are 24 simulated users. We warm up for 10,000 transactions, and we run for 3 checkpoints.
Java Server: SPECjbb2000 is a server-side java benchmark that models a 3-tier system. We used Sun's HotSpot 1.4.0 Server JVM. Our experiments use 1.5 threads/processor and 1.5 warehouses/processor (~500 MB of data). We warm up for 100,000 transactions, and we run for 3 checkpoints.
Static Web Server: We use Apache 2.0.36 for SPARC/Solaris 8, configured to use pthread locks and minimal logging as the web server. We use SURGE [3] to generate web requests. We use a repository of 20,000 files (totalling ~500 MB). There are 15 simulated users per processor. We warm up for ~80,000 requests, and we run for 3 checkpoints.
Dynamic Web Server: Slashcode is based on a dynamic web message posting system used by slashdot.org. We use Slashcode 2.0, Apache 1.3.20, and Apache's mod_perl 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of slashcode.com and it contains ~3,000 messages. A multithreaded driver simulates browsing and posting behavior for 12 users/processor. We warm up for 240 transactions, and we run for 3 checkpoints.

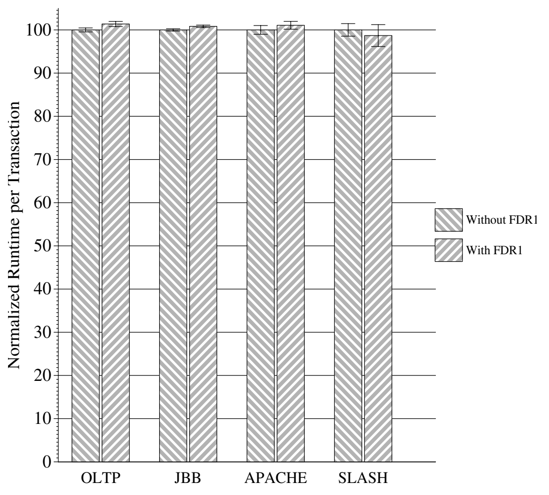

FIGURE 4. Performance impact of Flight Data Recorder. Obtained from 15 random runs of each workload.

gate the impact from the workload variabilities. Error bars in our results represent 95% confidence intervals of the mean.

We have argued that FDR algorithms are sufficient for deterministic replay. Not surprisingly, however, we found it necessary to debug the FDR1 implementation. We found several non-trivial bugs related to the protocol races with a random testing approach [28]. We developed a multi-threaded program whose final output is sensitive to the order of its frequent data races. In particular, the program computes a signature using a multiplicative congruential pseudo-random number generator [8]. We run the program on a memory system with pseudo-randomly perturbed latencies; each of ten thousands of runs produces a unique signature. If the replayed program computes a different signature, either the FDR implementation or the replayer is buggy. We have discovered several bugs through our random testing. This approach increases our confidence in the FDR1 implementation, but does *not* prove it correct.

8. Experimental Results

This section evaluates FDR1 on a four-processor system to show that the Flight Data Recorder is practical, not just theoretically possible.

Performance Impact. Figure 4 shows that FDR1 modestly affects program runtime. Specifically, it depicts benchmark runtimes (smaller is better) without and with FDR1 (normalized to the system without FDR1). The slowdown due to FDR1 overhead is less than 2% and not even statistically significant for Apache and Slash whose 95% confidence intervals overlap. This result is due to checkpoints being rare (so checkpoint overhead are paid only once every 333 million cycles) and, as is discussed below, the log bandwidth is sufficiently small (rarely causing processor stalls due to buffer overflow).

Importantly, results show that it is acceptable to leave FDR1 "always on" in anticipation of a trigger. We do not evaluate the time to create a log-enhanced core dump after a trigger, because this time is not critical in most applications of FDR.

Log Size. Figure 5 shows FDR1 log sizes. It plots log growth rate in MB/second/processor without (left) and with (right) compression. For these workloads, the log bandwidth requirements are acceptable and the physical memory needed to store a 1-second replay interval is mod-

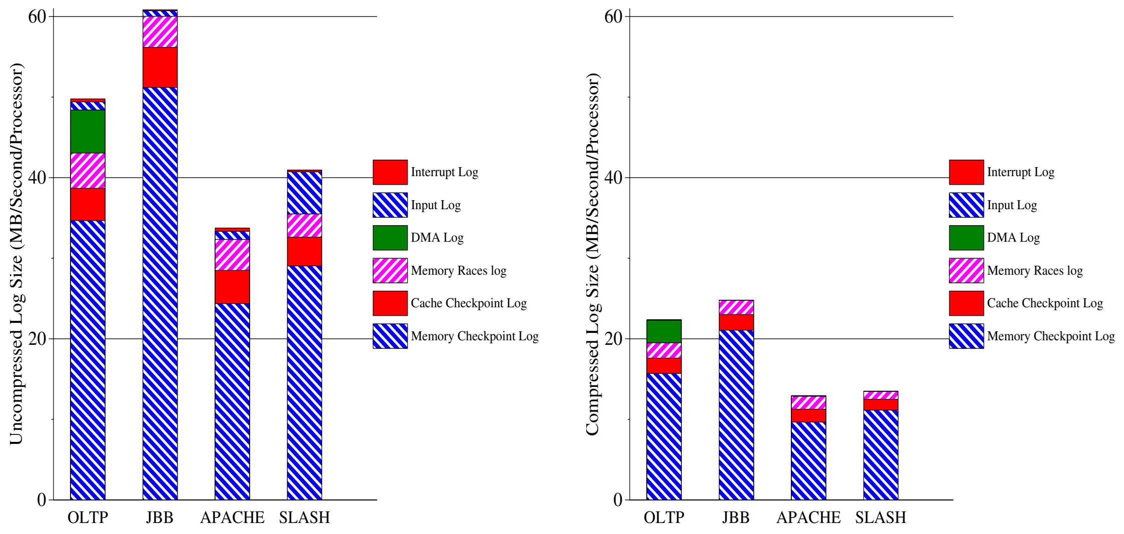

FIGURE 5. Log size breakdown of Flight Data Recorder. Data points are averaged from 15 runs of each workload.

est (e.g., 34 MB/processor for 1.33 second of the longest replay interval is less than 7% of a 512 MB/processor memory).

Figure 5 reveals two additional insights, via breaking logs into six parts: memory checkpoint log, cache checkpoint log, memory race log, DMA log, input log and interrupt log. First, memory and cache checkpoint logs dominate log size. This says that the race logging techniques developed in this paper work well and that future optimizations may focus on better checkpointing (e.g., by replacing SafetyNet). Second, except for OLTP, I/O log sizes are even smaller, in large part, because I/O is not frequent in these workloads.

Scaling. The above results showed that, for these workloads, FDR1's performance overhead and log sizes are reasonable. These workloads are much larger and do more I/O than the standard SPEC and SPLASH workloads. Nevertheless, real multiprocessor workloads may generate larger checkpoint logs (if worse locality adversely affects checkpoint log) and may do more I/O. Even the bad case of the sustained, peak input of a 1 Gb/s network interface, however, would increase uncompressed log storage by only 32 MB for each of the four processors. On the other hand, even today many multiprocessors ship with 1-2 GB of memory per processor, making even 100 MB per-processor logs viable. Moreover, there is nothing magical about our 1-second replay interval target. Larger workloads can target one-tenth of a second and still enable replays of the last 100 million cycles. Furthermore, alternative designs could use larger external storage (e.g. disks) to record much longer intervals or I/O intensive applications. Finally, we also found that FDR1 operates well in 16-processor systems.

9. Offline Replayer Discussion

A replayer can use the log-enhanced core dump provided by FDR to perform deterministic replay of the replay interval. A reasonable design lets an expert to debug using a high-level debugger that controls a low-level replayer.

The replayer would first initialize a core dump with the system state at the beginning of the replay interval. A replayer working with SafetyNet, for example, would select the first available register checkpoint and unroll memory logs. The replayer then replays the interval. It handles memory races by using log entries to constrain when it steps each processor (e.g., *j*'s log entry for i:34$\rightarrow$$j$:18 will stop *j* before instruction 18 until *i* completes instruction 34). The replayer also inserts I/O load values, interrupt values, and DMA store values at appropriate instruction counts. We developed a simple replayer for random testing (Section 7).

A full-function replayer must also provide the debugger with an interface to start, stop and step the execution, query and modify state, and other standard features of a modern debugger. We leave developing a full-function replayer for future work. One challenge is that the replayer enables access only to variables that reside in physical memory during the replay interval (because FDR1 does not currently capture disk state). State outside physical memory, however, cannot affect program execution until they are loaded into memory.

10. Conclusions

This paper develops a hardware technique, the *Flight Data Recorder*, for a multiprocessor system that continuously logs execution activity to enable full-system offline deterministic replay of a time interval preceding a software trigger. We develop a complete solution that logs all system activities, including OS and I/O. We show the Flight Data Recorder can be "always on" in anticipation of being triggered, because time and space overheads are modest. We present quantitative results for four commercial workloads and discuss how results scale with number of processors and memory size.

The Flight Data Recorder presented in the paper assumes sequential consistency implemented with directory cache coherence. Future work includes developing equally efficient recorders for other memory consistency models (e.g., Intel IA-32 processor consistency) and other coherence protocols (e.g., snooping). While the detailed hardware design may change, the approach of using a coherence protocol to trace memory races still applies.

11. Acknowledgements

We thank David Wood for recorder design suggestions, Peter Hsu and Bob Zak for early feedback, Eric Bach for suggesting the random number algorithm of the FDR tester, Milo Martin and Dan Sorin for invaluable simulator infrastructure assistance, and Carl Mauer for information on SPARC processor internals. We thank the following for presentation corrections and improvements: Alaa Alameldeen, Jarrod Lewis, Carrie Pritchard, Dan Sorin, and the Duke Architecture Reading Group. We thank the Multifacet project for a supportive environment.

References

[1] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[2] 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.

[3] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[4] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithread Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pages 48–59, Aug. 1998.

[5] A. Dinning and E. Schonberg. The Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–10, Mar. 1990.

[6] Geodesic Systems. *Geodesic TraceBack - Application Fault Management Monitor*. Geodesic Systems, Inc., 2003.

[7] D. Hunt and P. Marinos. A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems*, pages 170–175, 1987.

[8] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[9] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

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

[11] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[12] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, Feb. 2002.

[13] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.

[14] S. L. Min and J.-D. Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Apr. 1991.

[15] 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.

[16] R. H. B. Netzer and B. P. Miller. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[17] D. A. Patterson, et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical report, UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, 2002.

[18] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1996.

[19] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, May 2002.

[20] B. Richards and J. R. Larus. Protocol-based Data-race Detection. In *SIGMETRICS symposium on Parallel and Distributed Tools*, pages 40–47, 1998.

[21] M. Ronsse and K. D. Bosschere. Non-intrusive On-the-fly Data Race Detection using Execution Replay. In *Automated and Algorithmic Debugging*, Nov. 2000.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[23] C. E. Scheurich. Access Ordering and Coherence in Shared Memory Multiprocessors. Technical report, University of Southern California, Computer Engineering Division Technical Report No. CENG 89-19, May 1989.

[24] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, Apr. 1988.

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance. Technical Report 1420, Computer Sciences Department, University of Wisconsin–Madison, Oct. 2000.

[26] 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 *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[27] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research and Development*, 45(2):271–285, Mar. 2001.

[28] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design and Test of Computers*, pages 13–25, Aug. 1990.

[29] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.