# HARDWARE AND SOFTWARE APPROACHES FOR DETERMINISTIC MULTI-PROCESSOR REPLAY OF CONCURRENT PROGRAMS

## Contributors

**Gilles Pokam**
Intel Corporation

**Cristiano Pereira**
Intel Corporation

**Klaus Danne**
Intel Corporation

**Lynda Yang**
University of Illinois at Urbana-Champaign

**Sam King**
University of Illinois at Urbana-Champaign

**Josep Torrellas**
University of Illinois at Urbana-Champaign

## Index Words

Concurrent Programs
Deterministic Replay Debugging
Fault Tolerance
Non-determinism
Memory Race Recording
Chunks

## Abstract

As multi-processors become mainstream, software developers must harness the parallelism available in programs to keep up with multi-core performance. Writing parallel programs, however, is notoriously difficult, even for the most advanced programmers. The main reason for this lies in the non-deterministic nature of concurrent programs, which makes it very difficult to reproduce a program execution. As a result, reasoning about program behavior is challenging. For instance, debugging concurrent programs is known to be difficult because of the non-determinism of multi-threaded programs. Malicious code can hide behind non-determinism, making software vulnerabilities much more difficult to detect on multi-threaded programs.

In this article, we explore hardware and software avenues for improving the programmability of Intel® multi-processors. In particular, we investigate techniques for reproducing a non-deterministic program execution that can efficiently deal with the issues just mentioned. We identify the main challenges associated with these techniques, examine opportunities to overcome some of these challenges, and explore potential usage models of program execution reproducibility for debugging and fault tolerance of concurrent programs.

## Introduction

A common assumption of many application developers is that software behaves deterministically: given program A, running A on the same machine several times should produce the same outcome. This assumption is important for application performance, as it allows one to reason about program behavior. Most single-threaded programs executing on uni-processor systems exhibit this property because they are inherently sequential. However, when executed on multi-core processors, these programs need to be re-written to take advantage of all available computing resources to improve performance. Writing parallel programs, however, is a very difficult task because parallel programs tend to be non-deterministic by nature: running the same parallel program A on the same multi-core machine several times can potentially lead to different outcomes for each run. This makes both improving performance and reasoning about program behavior very challenging.

Deterministic multi-processor replay (DMR) can efficiently deal with the non-deterministic nature of parallel programs. The main idea behind DMR is reproducibility of program execution. Reproducing a multi-threaded program execution requires recording all sources of non-determinism, so that during replay, these threads can be re-synchronized in the same way as in the original execution. On modern chip multi-processor (CMP) systems, the sources of non-determinism can be either input non-determinism (data inputs, keyboard, interrupts, I/O, etc.) or memory non-determinism (access interleavings among threads). These sources of non-determinism can be recorded by using either software or hardware, or a combination of both.

Software-only implementations of DMR can run on legacy machines without hardware changes, but they suffer from performance slowdowns that can restrict the applicability of DMR. To achieve performance levels comparable to hardware schemes, software approaches can be backed up with hardware support. In this article, we describe what the software-only approaches for DMR may look like, and what types of hardware support may be required to mitigate their performance. Our discussion starts with the details of DMR: we focus on the usage models and on the main challenges associated with recording and replaying concurrent programs. We then describe several ways in which DMR schemes can be implemented in software, and we elaborate on the various tradeoffs associated with these approaches. Finally, we describe hardware extensions to software-only implementations that can help mitigate performance and improve the applicability of DMR.

*"Deterministic multi-processor replay (DMR) can efficiently deal with the non-deterministic nature of parallel programs."*

## Why Record-and-Replay Matters

Recording and deterministically replaying a program execution gives computer users the ability to travel backward in time, recreating past states and events in the computer. Time travel is achieved by recording key events when the software runs, and then restoring to a previous checkpoint and replaying the recorded log to force the software down the same execution path.

This mechanism enables a wide range of applications in modern systems, especially in multi-processor systems in which concurrent programs are subject to non-deterministic execution: such execution makes it very hard to reason about or reproduce a program behavior.

- *Debugging*. Programmers can use time travel to help debug programs [36, 39, 15, 4, 1] including programs with non-determinism [20, 33], since time travel can provide the illusion of reverse execution and reverse debugging.
- *Security*. System builders can use time travel to replay the past execution of applications looking for exploits of newly discovered vulnerabilities [19], to inspect the actions of an attacker [12], or to run expensive security checks in parallel with the primary computation [9].
- *Fault tolerance*. System designers can use replay as an efficient mechanism for recreating the state of a system after a crash [5].

*"Recording and deterministically replaying a program execution gives computer users the ability to travel backward in time."*

## Non-determinism of Concurrent Programs

*"Some of the input is not deterministic across different runs of the program, even if the program's command line arguments are the same."*

The goal of deterministic replay is to be able to reproduce the execution of a program in the way it was observed during recording. In order to reproduce an execution, each instruction should see the same input operands as in the original run. This should guarantee the same execution paths for each thread. During an execution, a program reads data from either memory or register values. Some of the input is not deterministic across different runs of the program, even if the program's command line arguments are the same. Hence, in order to guarantee determinism these inputs need to be recorded in a log and injected at replay. In this section, we describe these sources of non-determinism.

Deterministic replay can be done at different levels of the software stack. At the top level, one can replay only the user-level instructions that are executed. These include application code and system library code. This is the approach taken by BugNet [26], Capo [25], iDNA [3], and PinPlay [29]. At the lowest level, a system can record and replay all instructions executed in the machine, including both system-level and user-level instructions. Regardless of the level one is looking at, the sources of non-determinism can be divided into two sets: input read by the program and memory interleavings across different threads of execution. We now describe each source in more detail.

### Input Non-determinism

*"User-level replay has different requirements from those of system-level replay."*

Input non-determinism differs, depending on which layer of the system is being recorded for replay. User-level replay has different requirements from those of system-level replay. Conceptually, the non-deterministic inputs are all the inputs that are consumed by the system layer being recorded that are not produced by the same layer. For instance, for user-level replay, all inputs coming from the operating system are non-deterministic, because there is no guarantee of repeatability across two runs. A UNIX* system call, such as *gettimeofday*, is inherently non-deterministic across two runs, for instance. For a system-level record, all inputs that are external to the system are non-deterministic inputs. External inputs are inputs coming from external devices (I/O, interrupts, DMAs). We now discuss the source of non-determinism at each level.

*"For a system-level record, all inputs that are external to the system are non-deterministic inputs."*

For user-level replay, the sources of non-determinism are listed as follows:

- *System calls*. Many system calls are non-deterministic. An obvious example is a timing-dependent call, such as the UNIX call *gettimeofday*. Other system calls can also be non-deterministic. A system call reading information from a network card may return different results, or a system-call reading from a disk may return different results.
- *Signals*. Programs can receive asynchronous signals that can be delivered at different times across two runs, making the control flow non-deterministic.

- *Special architectural instructions*. On x86 architecture, some instructions are non-deterministic, such as RDTSC (read timestamp) and RDPMC (read performance counters). Across processor generations of the same architecture, CPUID will also return different values, if the replay happens in a processor other than the one in which the recording happened.

In addition to the non-deterministic inputs just mentioned, other sources of non-determinism at the user-level are the location of the program stack that can change from run to run and the locations where dynamic libraries are loaded during execution. Although these are not inputs to the program, they also change program behavior and need to be taken care of for deterministic replay.

*"Other sources of non-determinism at the user-level are the location of the program stack that can change from run to run and the locations where dynamic libraries are loaded during execution."*

At the system-level, the major sources of non-determinism are the following:

- *I/O*. It is common for most architectures to allow memory mapped I/O: loads and stores effectively read from and write to devices. If one is replaying the operating system code, the reads from I/O devices are not guaranteed to be repeatable. As a result, the values read by those load instructions need to be recorded.
- *Hardware interrupts*. Hardware interrupts trigger the execution of an interrupt service routine, which changes the control flow of the execution. Interrupts are used to notify the processor that some data (e.g., disk read) are available to be consumed. An interrupt is delivered at any point in time during the execution of the operating system code. A recorder needs to log the point at which the interrupt arrived and the content of the interrupt (what its source is: e.g., disk I/O, network I/O, timer interrupt, etc.).
- *Direct Memory Access (DMA)*. Direct memory accesses perform writes directly to memory without the intervention of the processor. The values written by DMA as well as the timestamp at which those values were written need to be recorded to be reproducible during replay.

In addition, the results of processor-specific instructions, such as x86 RDTSC, also need to be recorded as is the case with user-level code, in order to ensure repeatability.

*"A recorder needs to log the point at which the interrupt arrived and the content of the interrupt."*

*"In multi-core machines, an additional source of non-determinism is present and that is the order in which all threads in the system access shared memory."*

**Memory Interleaving**

Input non-determinism is present on single-core and multi-core machines. However, in multi-core machines, an additional source of non-determinism is present and that is the order in which all threads in the system access shared memory. This is typically known as memory races, where different runs of a program may result in different threads winning the race when trying to access a piece of shared memory. Memory races occur between synchronization operations (synchronization races) or between data accesses (data races). At the user-level, threads access memory in a different order, because the operating system may schedule them in a different order. This is due to interrupts being delivered at different times, because of differences in the architectural state (cache line misses, memory latencies, etc.) and also because of the load in the system. As a result, the shared memory values seen by each thread in different runs can change, resulting in different behavior for each thread across runs. This is the major source of non-determinism in multi-threaded programs. Races also occur among threads within the operating system, and the behavior across two runs is also not guaranteed to be the same. Hence the order in which races occur within the operating system code also needs to be recorded to guarantee deterministic replay.

*"The order in which races occur within the operating system code also needs to be recorded to guarantee deterministic replay."*

## Software Approaches for Deterministic Replay

Software-only approaches to record-and-replay (R&R) can be deployed on current commodity hardware at no cost. As described in the previous section, an R&R solution needs to tackle two issues: logging and replaying non-deterministic inputs and enforcing memory access interleavings. We describe software-only solutions to both of these challenges next, and we provide details on the techniques used in recent deterministic replay approaches extant in literature. Because there are more software-only R&R-like systems than can possibly be discussed in this article, we choose to mention those that best characterize our focus. Once we've surveyed the literature, we discuss the remaining open challenges in software-only solutions.

**Reproducing Input Non-determinism**

Systems and programs execute non-deterministically due to the external resources they are exposed to and the timing of these resources. Thus, these external resources can be all viewed as *inputs*, whether they are user inputs, interrupts, system call effects, etc. Given the same inputs and the same initial state, the behavior of the system or application is *deterministic*. The approach to R&R, therefore, is to log these inputs during the logging phase and inject them back during replay.

*"An R&R solution needs to tackle two issues: logging and replaying non-deterministic inputs and enforcing memory access interleavings."*

Table 1 summarizes the replay systems under discussion in terms of the level of replay (user-level or system-level), usage model, and how they are implemented for replaying inputs.

| Replay System | Level of Replay | Usage Model | Implementation |
|---|---|---|---|
| Bressoud and Schneider [5] | System | Fault-tolerance | Virtual machine |
| CapoOne [25] | User | General notion of "time travel" for multiple purposes | Kernel modifications, *ptrace* |
| Flashback [36] | User | Debugging | Kernel modifications |
| iDNA [3] | User | Debugging, profiling | Dynamic instrumentation |
| Jockey [34] | User | Debugging | Library-based, rewrites system calls |
| Liblog [16] | User | Debugging | Library-based, intercepts calls to *libc* |
| ODR [2] | User | Debugging | Kernel modifications, *ptrace* |
| PinPlay [29] | User | Debugging, profiling | Dynamic instrumentation |
| R2 [17] | User | Debugging | Library-based, stubs for replayed function calls |
| ReVirt [13] | System | Security | Virtual machine |
| TTVM [20] | System | Debugging | Virtual machine |
| VMWare [38] | System | General replay | Virtual machine |

**Table 1:** Summary of Approaches to Replaying Input Non-determinism
Source: Intel Corporation, 2009

**User-level Input Non-determinism**

First, let us consider user application replay. For the most part, we discuss how several approaches handle system calls and signals, since together they represent a large part of the non-deterministic external resources exposed to the application. They also represent resources that have inherently deterministic timing and non-deterministic timing, respectively.

*System Calls*

An application's interaction with the external system state is generally confined to its system calls. We discuss in detail how two recent replay systems — Flashback [36] and CapoOne [25] — handle these system calls. Flashback can roll back the memory state of a process to user-defined checkpoints, and it supports replay by logging the process's interaction with the system. Flashback's usage model is for debugging software. CapoOne can log and replay multiple threads and processes in a given replay group, cohesively, while concurrently supporting multiple independent replay groups. It re-executes the entire application during replay. CapoOne requires additional hardware to support multi-processor replay; however, its technique for enforcing an application's external inputs is completely software-based.

*"Flashback can roll back the memory state of a process to user-defined checkpoints."*

Both Flashback and CapoOne interpose on system call routines: they log the inputs, results, and side-effects (copy_to_user) of each system call, and they inject the data back in during re-execution of system call entry and exit points. If the effect of a given system call is isolated to only the user application (e.g., *getpid*()), the actual call is bypassed during replay, and its effects are emulated by injecting the values retrieved from the log. On the other hand, if a system call modifies a system state that is outside of the replayed application (e.g., *fork*()), the system call is re-executed during replay in a manner such that its effect on the application is the same as during the logging phase. CapoOne interposes on system calls in user space via the *ptrace* mechanism, while Flashback does so with kernel modifications. Another replay scheme called ODR [2] describes similar techniques to handle system calls, by using both *ptrace* and kernel modules. Jockey [34], a replay debugger, is slightly different from Flashback and CapoOne in that Jockey links its own shared-object file to the replayed application and then rewrites the system calls of interest.

*"Functions above the user-defined interface are re-executed during replay."*

While all of these approaches automatically define the interface at which logging and replay occur, namely the system call boundary, R2 [17] is a library-based replay debugger tool that allows the user to choose this demarcation. Functions above the user-defined interface are re-executed during replay, while those below it are emulated by using data from log files. Implementation-wise, R2 generates, and later calls, the stub associated with each function that needs to be logged or replayed. The authors of R2 also address the issue of preserving order between function calls that are executed by different threads. R2 uses a Lamport clock [21] to either serialize all calls or allow them to occur concurrently, as long as causal-order is maintained.

***Signals***

With system calls, we are only interested in recording their effects, since they always execute at the same point in a given application. This is, however, untrue for signals. The purpose of a signal is to notify an application of a given event, and since signals are asynchronous and can occur at any point during the application's execution, they are a good example of a non-deterministic input that is time-related. Although Flashback does not support signal replay, Flashback's developers suggest using the approach described in [35]: i.e., use the processor's instruction counter to log exactly when the signal occurred. During replay, the signal would be re-delivered when the instruction counter reaches the logged value. Jockey, on the other hand, delays all signals encountered during the logging phase until the end of the next system call, which it logs with that system call. Thus, during replay, the signal is re-delivered at the end of the same system call. CapoOne and liblog [16], another replay debugger, use a similar technique.

*"Since signals are asynchronous and can occur at any point during the application's execution, they are a good example of a non-deterministic input that is time-related."*

*Dynamic Instrumentation*

PinPlay [29] and iDNA [3] are replay systems that focus on the application debugging usage model: they are based on the dynamic binary instrumentation of a captured program trace. Non-deterministic input is logged and replayed, by tracking and restoring changes to registers and main memory. PinPlay replays asynchronous signals by logging the instruction count of where signals occur.

*"Non-deterministic input is logged and replayed, by tracking and restoring changes to registers and main memory."*

**Full-system Input Non-determinism**

We move on to consider approaches for software-based, full-system replay, which include ReVirt [13], TTVM [20], the system described by [5], and VMWare [38]. The first three were designed for the usage models of security, debugging, and fault tolerance. Perhaps, unsurprisingly, all of these methods take advantage of virtual machines.

ReVirt uses UMLinux [6], a virtual machine that runs as a process on the host. Hardware components and events of the guest are emulated by software analogues. For example, the guest hard disk is a host file, the guest CD-ROM is a host device, and guest hardware interrupt events are simulated by the host delivering a signal to the guest kernel. With these abstractions, ReVirt is able to provide deterministic replay by checkpointing the virtual disk and then logging and replaying the inputs that are external to the virtual machine. Similar to user-application replay, each external input may require that only the data associated with it need be logged, or additionally, it may require that a timing-factor for those that are asynchronous be logged as well. ReVirt logs the input from external devices such as the keyboard and CD-ROM, non-deterministic results returned by system calls from the guest kernel to the host kernel, and non-deterministic hardware instructions such as *RDTSC*. Guest hardware interrupts, emulated by signals, are asynchronous, and thus ReVirt has to ensure that these are delivered at the same point in the execution path. The authors chose to use the program counter and the hardware retired branches counter to uniquely identify the point to deliver the signal.

*"ReVirt is able to provide deterministic replay by checkpointing the virtual disk and then logging and replaying the inputs that are external to the virtual machine."*

TTVM uses ReVirt for its logging and replaying functionality, but makes changes that make it more suitable for its debugging usage model; for example, TTVM provides support for greater and more frequent checkpoints.

**Reproducing Memory Access Non-determinism**

The techniques we just described guarantee determinism for replaying single-threaded applications or multi-threaded applications where the threads are independent from one another. Deterministic replay of multi-threaded applications, with threads communicating via synchronization or through shared memory, require additional support.

*"Deterministic replay of multi-threaded applications, with threads communicating via synchronization or through shared memory, require additional support."*

Table 2 summarizes the replay systems we describe next in terms of usage model, multi-processor support, support for replaying data-races without additional analysis, and support for immediate replay without a state-exploration stage.

| Replay System | Usage Model | Multiprocessor Support? | Data Race Support? | Immediate Replay (no offline state-exploration stage)? |
|---|---|---|---|---|
| DejaVu [8] | Debugging | No | Yes | Yes |
| iDNA [3] | Debugging, profiling | Yes | No | Yes |
| Instant Replay [23] | Debugging | Yes | No | Yes |
| Kendo [27] | Debugging, fault-tolerance | Yes | No | Yes |
| Liblog [16] | Debugging | No | Yes | Yes |
| ODR [2] | Debugging | Yes | Yes | No |
| PinPlay [29] | Debugging | Yes | Yes | Yes |
| PRES [28] | Debugging | Yes | Yes | No |
| RecPlay [32] | Debugging | Yes | No | Yes |
| Russinovich and Cogswell [33] | Debugging | No | Yes | Yes |
| SMP-ReVirt [11] | General replay | Yes | Yes | Yes |

**Table 2:** Summary of Approaches to Replaying Memory Access Non-determinism
Source: Intel Corporation, 2009

**Replay in Uniprocessors**

In a uni-processor system, it was observed that since only one thread can run at any given time, recording the order of how the threads were scheduled on the processor is sufficient for later replaying of the memory access interleaving [16, 8, 33]. These solutions have been implemented at the operating-system level [33], virtual-machine level [8], and user level [16].

*"On a multi-processor, thread-scheduling information is not sufficient for deterministic replay, since different threads can be running on different processors or cores concurrently."*

**Replay of Synchronized Accesses**

On a multi-processor, thread-scheduling information is not sufficient for deterministic replay, since different threads can be running on different processors or cores concurrently. Earlier proposals, such as Instant Replay [23] and RecPlay [32], recorded the order of operations at a coarse granularity; that is, at the level of user-annotated shared objects and synchronization operations, respectively. Therefore, these schemes were only able to guarantee deterministic replay for data-race free programs. Both proposals were designed with debugging in mind. As an illustrative example, Instant Replay used the concurrent-read-exclusive-write (CREW) [10] protocol when different threads wanted access to a shared object. CREW guarantees that when a thread has permission to write to a shared object, no other threads are allowed to write to or read from that object. On the other hand, multiple threads can read from the object concurrently. Instant Replay uses the recorded sequence of write operations and the "version" number of the object for each read operation during replay.

Some recent proposals also do not support deterministic replay of programs with data races. iDNA [3] schedules a thread's execution trace according to instruction sequences that are ordered via synchronization operations. Kendo [27] offers deterministic multi-threading in software by assuring the same sequence of lock acquisition for a given input. While not technically a replay system, Kendo also requires that programs be correctly synchronized. Kendo's usage models include debugging and support for fault-tolerant replicas.

*"iDNA [3] schedules a thread's execution trace according to instruction sequences that are ordered via synchronization operations."*

**Replay with State-exploration**

ODR [2] and PRES [28] are two novel approaches that facilitate replay debugging, but are not able to immediately replay an application, given the log data during the logging phase. Instead, they intelligently explore the space of possible program execution paths until the original output or bug is reproduced. Such analysis must be done off-line, but ODR and PRES gain in having smaller logging phase overtimes (since they log less data) compared to software schemes that provide for immediate replay.

**PinPlay and SMP-ReVirt**

PinPlay [29] and SMP-ReVirt [11, 14] provide for immediate replay, and they order shared memory operations rather than coarse-grained objects.

PinPlay's approach is to implement a software version of the flight data recorder (FDR) [37]. FDR exploits cache coherence messages to find memory access dependencies and to order pairs of instructions.

*"FDR exploits cache coherence messages to find memory access dependencies and to order pairs of instructions."*

SMP-ReVirt is a generalization of the CREW protocol for shared objects in Instant Replay [23] to shared pages of memory. A given page in memory can only be in a state that is concurrently read or exclusively written during the logging phase. These access controls are implemented by changing a thread's page permissions — read-access, write-access, or no-access for a given page — during the system's execution. For example, if a thread wants to write to a page and thus needs to elevate its permission to write-access, all other threads must have their permissions reduced to no-access first. Each thread has its own log. When a thread has its page permission elevated during logging, it logs the point at which it received the elevated permission and the points where the other threads reduced their page permissions. Additionally, the threads that had their permissions reduced log the same points where their permissions were reduced. SMP-ReVirt specifies these "points" in the execution of the system by means of instruction counts. The instructions count of each processor is also updated in a globally visible vector. Thus, during replay, when a thread encounters a page permission elevation entry, it waits until the other permission-reducing threads reach the instruction count value indicated in the log. On the other hand, when a thread encounters a page permission reduction entry, it updates the global vector with its instruction count.

*"Enforcing input determinism in software seems to be a reasonable approach, considering the low overhead."*

**Challenges in Software-only Deterministic Replay**

It has been shown that designing a software-only solution for recording and replaying input non-determinism is reasonable in terms of execution speed, and it can be done with an overhead of less than ten percent [20, 28, 36]. It is difficult to compare and summarize input log size growth rates for the different approaches discussed here, since different approaches log different events, may compress the log differently, and use different applications as their benchmarks. However, it can be noted that Flashback's [36] log size is linear to the number of system call invocations. Other similar input logging techniques may likely exhibit similar behavior. In short, enforcing input determinism in software seems to be a reasonable approach, considering the low overhead.

Conversely, the overhead incurred in enforcing memory access interleaving in software is a different story. SMP-ReVirt [11, 14] and PinPlay [29] allow for the most flexible and immediate replay, but they incur a huge overhead. Since SMP-ReVirt instruments and protects shared memory at the page level of granularity, it has issues with false sharing and page contention [28], especially as the number of processors increases [14]. With four CPUs, the logging phase runtime of an application in SMP-ReVirt can be up to 9 times that of a native run [14]. PinPlay, like iDNA, which uses dynamic instrumentation and has a 12 to 17 times slowdown [3], cannot be turned on all the time.

*"Another challenge with software-based schemes is their ability to pinpoint asynchronous events during replay."*

The rest of the schemes previously described for replaying multi-threaded applications are either less flexible (uniprocessor only [8, 16, 33], data-race free programs only [3, 23, 27, 32]), or they trade off short on-line recording times with potentially long off-line state exploration times for replay [2, 28].

Another challenge with software-based schemes is their ability to pinpoint asynchronous events during replay. This issue was exemplified earlier in reference to asynchronous signals and interrupts. While some replay schemes choose to use hardware performance counters in their implementation [36, 35, 13], others choose to delay the event until a later synchronous event occurs [25, 34, 16]. The latter solution, though simpler, can theoretically affect program correctness, while the former solution requires the use of performance counters that are often inaccurate and non-deterministic [11, 27].

*"In the end, the selection of an appropriate replay system depends on the usage model."*

In the end, the selection of an appropriate replay system depends on the usage model. If we are to assume a debugging model where a programmer may not mind waiting a while for a bug to be reproduced, large replay overheads, though not desirable, may be reasonable. In fact, for most of the methods described here, the developers assumed a debugging usage model. Alternatively, a fault-tolerance replay model would require that backup replicas be able to keep up with the production replica, and thus good performance would be much more important. Note that performance is not the only factor that should be considered when determining which replay system works best with a usage model. For example, if the usage model is to replay system intrusions, it would be more suitable to use a full-system replay scheme rather than a user-application replay scheme.

## Hardware Support for Recording Memory Non-determinism

Deterministically replaying a program execution is a very difficult problem, as we just described. In addition to logging input non-determinism, existing software approaches have to record the interleavings of shared memory accesses. This can be done at various levels of granularity (e.g., page level or individual memory operations), but as discussed previously, the overhead incurred can be prohibitive and therefore detrimental to applications of R&R, such as fault-tolerance. For this reason, there has been a lot of emphasis on providing hardware support for logging the interleavings of shared memory accesses more efficiently. We call the proposed mechanisms for logging the order in which memory operations interleave memory race recorders (MRR).

*"There has been a lot of emphasis on providing hardware support for logging the interleavings of shared memory accesses more efficiently."*

Prior work on hardware support for MRR piggybacks on timestamps located on cache coherence messages and logs the outcome of memory races by using either a *point-to-point* or a *chunk-based* approach. In this section we describe these two approaches and suggest directions for making them practical in modern multi-processor systems.

### Point-to-point Approach

In point-to-point approaches [26, 37], memory dependencies are tracked at the granularity level of individual shared memory operations. In this approach, each memory block has a timestamp, and each memory operation updates the timestamp of the accessed block. In general, a block can be anything ranging from a memory word to multiple memory words [37, 31]. We now describe the FDR [37], a state-of-the-art implementation of a point-to-point MRR approach.

FDR augments each core in a multi-processor system with an instruction counter (IC) that counts retired instructions. FDR further augments each cache line with a cache instruction count (CIC) that stores the IC of the last store or load instruction that accessed the cache line (see Figure 1). When a core receives a remote coherence request to a cache line, it includes the corresponding CIC and its core ID in the response message. The requesting core can then log a dependency by storing the ID and CIC of the responding core and the current IC of the requesting core. To reduce the amount of information logged by the requesting core, a dependency is logged only if it cannot be inferred by a previous one. This optimization is called *transitive reduction*. For example, in Figure 1, only the dependency from *T1:W(b) to T2:R(b)* is logged, as *T1:R(a) to T2:W(a)* is consequentially implied by *T1:W(b) to T2:R(b)*. Transitive reduction is implemented by augmenting each core with a vector instruction count that keeps track of the latest CIC received by each core.

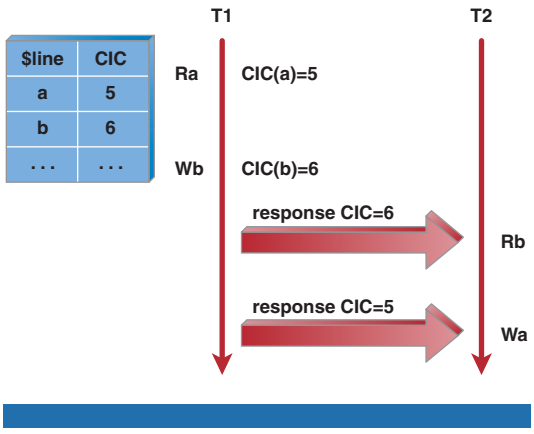


**Figure 1:** Point-to-point Approach
Source: Intel Corporation, 2009

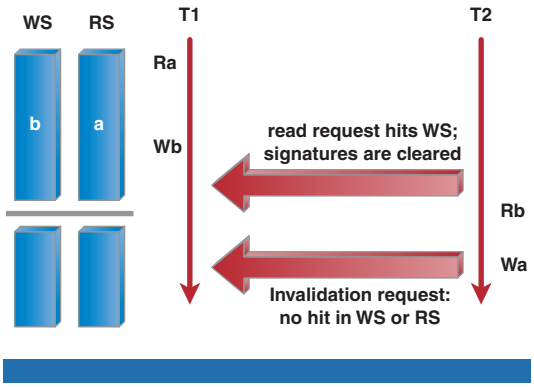


**Figure 2:** Chunk-based Approach
Source: Intel Corporation, 2009

## Chunk-based Approach

A chunk defines a block of memory instructions that executes in isolation, i.e., without a remote coherence request intervening and causing a conflict. Chunks are represented by using signatures, which are hardware implementations of Bloom Filters. Signatures are used to compactly represent sets of locally accessed read or write memory addresses and to disambiguate a remote shared memory reference against them. A conflict with the signatures ends a chunk and clears the signatures.

Similar to point-to-point approaches, chunk-based approaches can also take advantage of transitive reduction to reduce the amount of logged information. As shown in Figure 2, the remote read *T2:R(b)* conflicts with the write signature of T1 and causes T1 to end its chunk and to clear its signatures. Consequently, the request *T2:W(a)* does not conflict and the dependency *T1:R(a) to T2:W(a)* is implied. In contrast to point-to-point approaches in which a timestamp is stored with each memory block, chunk-based approaches only need to store a timestamp per core to order chunks between threads.

We now describe two similar implementations of chunk-based approaches.

### Rerun

In Rerun [18], episodes are like chunks. Rerun records a memory dependency by logging the length of an episode along with a timestamp. To identify the episodes that need to be logged, Rerun augments each core with a read and a write signature that keep track of the cache lines read and written by that core during the current episode, respectively. When a cache receives a remote coherence request, it checks its signatures to detect a conflict. If a conflict is detected, the core ends its current episode, which involves clearing the read and write signatures, creating a log entry containing the length of the terminating episode along with its timestamp, and updating the timestamp value. The timestamp represents a scalar clock maintained by each core to provide a total order among the episodes. The cores keep their timestamp up to date by piggybacking them on each cache coherence reply.

*"If a conflict is detected with a chunk, its signatures are cleared and the chunk is squashed and re-executed."*

Deterministic replay is achieved by sequentially executing the episodes in order of increasing timestamps. To do so, a replayer typically examines the logs to identify which thread should be dispatched next and how many instructions it is allowed to execute until an episode of a different thread needs to be replayed.

### DeLorean

Similar to Rerun, DeLorean [24] also logs chunks by using signatures, but does so in a different multi-processor execution environment. In this environment, cores continuously execute chunks that are separated by register checkpoints. A chunk execution in this environment is speculative, i.e., its side effects are visible to other cores only until after commit. Before a chunk can commit, however, its signatures are compared against the signatures of other chunks to detect conflicts. If a conflict is detected with a chunk, its signatures are cleared and the chunk is squashed and re-executed. While such an execution environment is not standard in today's multi-processors, it has been shown to perform well [7]. The required hardware extensions are similar to hardware-supported transactional memory systems [22].

To enable deterministic replay, DeLorean logs the total order of chunk commits. Because in this execution environment chunks have a fixed size, e.g., 1000 dynamic instructions, no additional information needs to be logged, except in the rare cases where chunks need to end early because of events such as interrupts. Consequently, the log size of DeLorean is about one order of magnitude smaller than in Rerun. DeLorean can even reduce the log size by another order of magnitude when operating in *PicoLog* mode. In this execution mode, the architecture commits chunks in a deterministic order, e.g., round robin. Although this execution mode sacrifices performance, DeLorean only needs to log the chunks that end due to non-deterministic events, such as interrupts.

*"The log size of DeLorean is about one order of magnitude smaller than in Rerun."*

**Making Memory Race Recorders Practical for Modern CMPs**

MRR approaches discussed so far are effective in logging the events required for deterministic replay, but they also impose some non-negligible amount of complexity and performance cost that can preclude hardware vendors from deploying similar solutions in real products. In this section, we pinpoint some of these issues and discuss possible ways to remedy them.

**Implementation Complexity**

Showstoppers with previous MRR approaches are the implementation complexity of proposed techniques and their associated hardware cost.

*"Showstoppers with previous MRR approaches are the implementation complexity of proposed techniques and their associated hardware cost."*

With point-to-point approaches, for instance, a hardware estate for storing the timestamp of each accessed memory block is required. If the granularity of a memory block is a cache line, then each cache line must be augmented with storage for the timestamp. Because a cache line eviction throws away the information stored into it, the timestamp must also be stored at the next cache level to reduce logging frequency. FDR estimates this cost to be ~6 percent of the capacity of a 32KB L1 cache.

The main hardware cost associated with chunk-based approaches lies in the storage required for signatures. In Rerun, for instance, the authors suggest using 1024-bit signatures to store read memory addresses and 256-bit signatures to store written memory addresses. In contrast to point-to-point approaches, these changes do not require modifications to the cache sub-system and are therefore less invasive. However, there is some complexity involved in implementing signatures. The authors in [30] show that implementing signatures in modern CMPs involves subtle interactions with the hierarchy and policy decisions of on-chip caches. The authors show that the signature placement in a multi-level cache hierarchy can degrade performance by increasing the traffic to the caches. They propose hybrid L1/L2 signature placement strategies to mitigate this performance degradation.

*"The main hardware cost associated with chunk-based approaches lies in the storage required for signatures."*

**Performance Overhead**

With the exception of DeLorean, all MRR approaches discussed in this section must piggyback on cache coherence messages to maintain ordering among events in the system. For instance, in FDR, the core ID and the CIC are piggybacked on each coherence reply to log a point-to-point dependency between two instructions from different threads, whereas in Rerun a timestamp is piggybacked on each coherence reply to maintain causal ordering between chunks. This overhead can hurt performance by putting a lot of pressure on the bandwidth. FDR and Rerun, for instance, report a performance cost of ~10 percent, an estimation based on functional simulation. Ideally, we want this coherence traffic overhead to be nonexistent in real implementations of MRR. One way to attain this objective with a chunk-based approach has recently been proposed in [30]. The authors make the observation that maintaining causality at the chunk boundary is all that is needed to order chunks. Doing so eliminates the requirement to piggyback a timestamp on each coherence message. Using this approach, they show that the coherence traffic overhead can be reduced by several orders of magnitude compared to Rerun or FDR.

*"The authors make the observation that maintaining causality at the chunk boundary is all that is needed to order chunks."*

*"DeLorean and FDR can replay a program at production run speed."*

**Replay Performance**

As discussed previously, there are plenty of applications that can benefit from deterministic replay. Each of these applications places different replay speed requirements on the system. For instance, while an application developer can easily accommodate slow replay during debugging, this is not the case for high-availability applications in which the downtime window during recovery must be shortened. Slow replay in this case can have devastating effects on the system. Instead, we would like a second machine to continuously replay the execution of the primary machine at a similar speed, and to be able to take over instantly if the primary fails. Ideally, we do not want a R&R system to be constrained by speed, because such a constraint would limit the system's scope and restrict its applicability. Therefore, techniques are needed to improve the replay speed of MRR approaches.

DeLorean and FDR can replay a program at production run speed. In DeLorean, this is achieved by replaying chunks in parallel and re-synchronizing them according to the same commit order as recorded during their original execution. With FDR, threads are replayed in parallel and are only re-synchronized at the locations corresponding to the point-to-point dependencies recorded during their original execution. Neither FDR nor DeLorean, however, is a likely choice for a practical MRR implementation today. As discussed previously, the complexity of FDR is a major showstopper in modern CMPs. For DeLorean, the execution environment it assumes is not standard in today's multi-processors.

Replaying episodes in Rerun is done sequentially, following increasing timestamp order. As such, Rerun cannot therefore meet the replay speed requirement of DMR usage models, such as fault-tolerance. As an alternative to Rerun, the authors in [30] have proposed a chunk-based replay scheme called a concurrent chunk region. A concurrent chunk region defines a set of chunks that can be replayed in parallel, because each chunk in such a region features the same timestamp as the chunk in other regions. To build such concurrent chunk regions, whenever a chunk must terminate due to a conflict, for instance, all chunks with similar timestamps must also be terminated simultaneously. Therefore, concurrent chunk regions trade off replay speed for log size. The authors in [30] have shown that, by using concurrent chunk regions, replay speed can be improved by several orders of magnitude at the cost of moderate log size increases.

*"A concurrent chunk region defines a set of chunks that can be replayed in parallel."*

## Conclusions

In this article we presented a comprehensive survey of DMR techniques to deal with multi-threaded program execution on CMP machines. We showed that software-only implementations of DMR are quite effective in recording and replaying concurrent programs, but they suffer from performance limitations that can restrict their applicability. To improve on performance, the memory non-determinism of multi-threaded programs must be recorded more efficiently. We described the hardware support needed to deal with fine-grained logging of memory interleavings more efficiently, using either point-to-point approaches or chunk-based approaches. Combined with software approaches, these hardware techniques can provide better performance and address a wider range of usage models. However, there are still several remaining challenges that need to be met before a complete solution can be deployed on real hardware. One such challenge involves recording memory non-determinism with non-sequentially consistent memory models. We hope that the discussions presented here help foster the research on DMR and that they stimulate a broader interest in DMR usage models.

*"To improve on performance, the memory non-determinism of multi-threaded programs must be recorded more efficiently.."*

## References


[1] H. Agrawal. "Towards automatic debugging of computer programs." PhD thesis, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, 1991.

[2] G. Altekar and I. Stoica. "ODR: Output-deterministic replay for multicore debugging." In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 193-206, 2009.

[3] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. "Framework for instruction-level tracing and analysis of program executions." In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154-163, 2006.

[4] B. Boothe. "A fully capable bidirectional debugger." *ACM SIGSOFT Software Engineering Notes*, 25(1), pages 36-37, 2000.

[5] T. C. Bressoud and F. B. Schneider. "Hypervisor-based fault tolerance." In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1-11, 2009.

[6] K. Buchacker and V. Sieh. "Framework for testing the fault-tolerance of systems including OS and network aspects." In *Proceedings of the 6th IEEE International Symposium on High-Assurance Systems Engineering: Special Topic: Impact of Networking*, pages 95-105, 2001.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. "BulkSC: Bulk enforcement of sequential consistency." *ACM SIGARCH Computer Architecture News*, 35(2), pages 278-289, 2007.

[8] J. Choi and H. Srinivasan. "Deterministic replay of Java multithreaded applications." In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48-59, 1998.

[9] J. Chow, T. Garfinkel, and P. M. Chen. "Decoupling dynamic program analysis from execution in virtual environments." In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2008.

[10] P. Courtois, F. Heymans, and D. Parnas. "Concurrent control with readers and writers." *Communications of the ACM*, 14(10), pages 667-668, 1971.

[11] G. Dunlap. "Execution replay for intrusion analysis." PhD thesis, EECS Department, University of Michigan, Ann Arbor, Michigan, 2006. Available at ***http://www.eecs.umich.edu/~pmchen/papers/dunlap06.pdf***

[12] S. King, G. Dunlap, and P. Chen. "Operating system support for virtual machines." In *Proceedings of the 2003 USENIX Technical Conference*, pages 71-84, 2003.

[13] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay." *ACM SIGOPS Operating Systems Review*, 36(SI), pages 211-224, 2002.

[14] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. "Execution replay of multiprocessor virtual machines." In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121-130, 2008.

[15] S. Feldman and C. Brown. "IGOR: a system for program debugging via reversible execution." In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112-123, 1988.

[16] D. Geels, G. Altekar, S. Shenker, and I. Stoica. "Replay debugging for distributed applications." In *Proceedings of the USENIX '06 Annual Technical Conference*, 27, 2006.

[17] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, F. Kaashoek, and Z. Zhang. "R2: An application-level kernel for record and replay." In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, pages 193-208, 2008.

[18] D. Hower and M. Hill. "Rerun: Exploiting episodes for lightweight memory race recording." *ACM SIGARCH Computer Architecture News*, 36(3), pages 265-276, 2008.

[19] A. Joshi, S. King, G. Dunlap, and P. Chen. "Detecting past and present intrusions through vulnerability-specific predicates." In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 91-104, 2005.

[20] S. King, G. W. Dunlap, and P. M. Chen. "Debugging operating systems with time-traveling virtual machines." In *Proceedings of the USENIX Annual Technical Conference*, 1, 2005.

[21] L. Lamport. "Time, clocks and the ordering of events in a distributed system." *CACM*, 21(7):558–565, 1978.

[22] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[23] T. LeBlanc and J. Mellor-Crummey. "Debugging parallel programs with instant replay." *IEEE Transactions on Computers*, 36(4), pages 471-482, 1987.

[24] P. Montesinos, L. Ceze, and J. Torrellas. "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently." In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 289-300, 2008.

[25] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. "Capo: Abstractions and software-hardware interface for hardware-assisted deterministic multiprocessor replay." In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 73-84, 2009.

[26] 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, 2005.

[27] M. Olszewski, J. Ansel, and S. Amarasinghe. "Kendo: Efficient deterministic multithreading in software." In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97-108, 2009.

[28] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. "PRES: Probabilistic replay with execution sketching on multiprocessors." In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 177-192, 2009.

[29] C. Pereira. "Reproducible user-level simulation of multi-threaded workloads." PhD thesis, Department of Computer Science and Engineering, University of California – San Diego, San Diego, California, 2007. Available at ***http://cseweb.ucsd.edu/~calder/papers/thesis-cristiano.pdf***.

[30] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A. Adl-Tabatabai. "Architecting a chunk-based memory race recorder in modern CMPs." In *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

[31] M. Prvulovic. "CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection." In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, 2006.

[32] M. Ronsse and K. De Bosschere. "RecPlay: a fully integrated practical record/replay system." *ACM Transactions on Computer Systems*, 17(2), pages 133–152, 1999.

[33] M. Russinovich and B. Cogswell. "Replay for concurrent non-deterministic shared-memory applications." In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming language Design and Implementation*, pages 258-266, 1996.

[34] Y. Saito. "Jockey: A user-space library for record-replay debugging." In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, pages 69-76, 2005.

[35] J. Slye and E. Elnozahy. "Supporting nondeterministic execution in fault-tolerant systems." In *Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, pages 250, 1996.

[36] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. "Flashback: A lightweight extension for rollback and deterministic replay for software debugging." In *Proceedings of the USENIX Annual Technical Conference*, 3, 2004.

[37] M. Xu, R. Bodik, and M. Hill. "A flight data recorder for enabling full-system multiprocessor deterministic replay." In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122-135, 2003.

[38] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and VMWare Inc. "Retrace: Collecting execution trace with virtual machine deterministic replay." In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[39] M. Zelkowitz. "Reversible execution." *Communications of the ACM*, 16(9):566, 1973.

## Authors' Biographies

**Gilles Pokam** is a Senior Research Scientist in the Microprocessor & Programming Research at Intel Labs. His research interests are in multi-core architectures and software, with a current focus on concurrent programming and programmer productivity. Before joining Intel Labs, he was a researcher at IBM T.J. Watson Research Center in NY, and a postdoctoral research scientist at the University of California, San Diego. He received a PhD degree from INRIA Lab and the University of Rennes I, in France. He is the recipient of the IEEE MICRO Top Picks Award 2006 that recognizes the most significant papers in computer architecture. Gilles is a member of IEEE and ACM. His e-mail is gilles.a.pokam at intel.com.

**Cristiano Pereira** is an Engineer in the Technology Pathfinding and Innovation team, at Intel's Software and Services Group. His research interests are in the areas of hardware support for better programmability of multi-core architectures and software tools to improve programmer's productivity. He received a PhD degree from the University of California, San Diego, in 2007 and a Masters degree from the Federal University of Minas Gerais, Brazil, in 2000. Prior to that, Cristiano worked for a number of small companies in Brazil. He is a member of IEEE. His e-mail is cristiano.l.pereira at intel.com.

**Klaus Danne** is an Engineer in the Microprocessor & Programming Research Group at Intel Labs. His research interests are in multi-core architectures, deterministic replay, design emulation, and reconfigurable computing systems. He received a PhD degree and Masters degree from the University of Paderborn Germany in 2006 and 2002, respectively. His e-mail is klaus.danne at intel.com.

**Lynda Yang** is a graduate student in Computer Science at the University of Illinois at Urbana-Champaign. Her research interests are in multi-processor architectures and operating systems. She received a BS degree in Computer Science in 2008 from the University of North Carolina at Chapel Hill. Her e-mail is yang61 at illinois.edu.

**Samuel T. King** is an Assistant Professor in the Computer Science Department at the University of Illinois. His research interests include security, experimental software systems, operating systems, and computer architecture. His current research focuses on defending against malicious hardware, deterministic replay, designing and implementing secure web browsers, and applying machine learning to systems problems. Sam received his PhD degree in Computer Science and Engineering from the University of Michigan in 2006.

**Josep Torrellas** is a Professor of Computer Science and Willett Faculty Scholar at the University of Illinois, Urbana-Champaign. He received a PhD degree from Stanford University in 1992. His research area is multi-processor computer architecture. He has participated in the Stanford DASH and the Illinois Cedar experimental multi-processor projects, and in several DARPA initiatives in novel computer architectures. Currently, he leads the Bulk Multi-core Architecture project for programmability in collaboration with Intel. He has published over 150 papers in computer architecture and received several best-paper awards. He has graduated 27 PhD students, many of whom are now leaders in academia and industry. He is an IEEE Fellow.

## Copyright