A Survey in Deterministic Replaying Approaches in Multiprocessors

Ahmad Heydari, Saeed Azimi

Abstract — most multithread executions acts nondeterministic while execute on multiprocessor systems. Recording nondeterministic events in such executions can enable deterministic replay. So some researchers are focused on deterministic replay capability in recording important information during executions.

In this survey we introduce Hardware-Base, Software-Base and Software-Hardware approaches that have been proposed for deterministic replaying. The Hardware-Base approaches, adds special hardware to detect the interleaving of memory accesses or instructions from the different processors during execution, and saw the key information in a log. This log is later used to replay the same interleaving. The Software-Base approaches propose software systems that record all sources of non-determinism, such as network inputs or interrupts, and use this information to guide the software down the same execution path during replay. Finally we introduce the Software-Hardware approaches with name Capo [25]. This method in hardware, records interleaving of the R-threads running in a log within the same sphere. And in software records the other sources of non-determinism that may affect the execution path of the R-threads, such as system call return values and signals.

Index Term — Deterministic Replay, Execution, Logging, Multiprocessor, Recording.

I. INTRODUCTION

Modern computer systems are nondeterministic due to a Variety of events that occur during an execution, including I/O, interrupts, and DMA fills, and interrupts, are relatively rare events. Deterministic replay of a uniprocessor machine has already proven useful for debugging [33] and intrusion detection [31] applications. Unfortunately current systems record the last state of the system and this makes cause of system failures very challenging. So some researchers are focused on deterministic replay capability in recording important information during execution. Deterministic replay is useful for several current and future applications. Some of these applications are:

- Cyclic Debugging: Deterministic replay is a process with a repeatedly capability in order to determine the cause or failure. Without Cyclic Debugging, a developer may not be able to effectively debug her program, because bugs may not faithfully reappear even in a controlled debugging environment [2], [15]. Debugging multi thread codes are challenging. Because concurrency bugs usually operates in specific conditions and their effects often clear after a few instructions. Therefore finding effective debugging techniques is crucial. Recording some logs like number of memory access during multi thread execution can be useful in replaying.

- Security: Deterministic replay could also be used to enhance the security of software by providing the means for an in depth analysis of an attack, hopefully leading to rapid Patch deployment and a reduction in the economic impact of new threats [2].

- Fault Tolerance: With the ability to replay an execution, it may also be possible to develop hot-standby systems for critical service providers using commodity hardware [2]. Deterministic replay can be used in fault tolerance in two ways: (1) replay-based redundancy and (2) replay-based failure recovery [15].

- Data Recovery: In the future, deterministic replay can be used as a data recovery method, because data can be observed and retrieved from a replay. For example, when a word processing program is replayed, a user can recover the text he typed into the editor but never saved [15]. Deterministic replay is divided into two phases [15]: the recording phase and the replay phase. In the recording phase, necessary information about the original execution is recorded. This information is then used in the replay phase to make the replay execution logically equivalent to the original execution.

Ahmad Heydari is with the Computer Engineering Department, Islamic Azad University Zanjan Branch, Iran. (e-mail: a.heydari@engineer.com).

Saeed Azimi is with the Computer Engineering Department, Islamic Azad University Zanjan Branch, Iran. (e-mail: s.azimi63@gmail.com).
software-only solutions do not perform well on workloads that interact frequently [2]. Thus, it is likely that a general solution will require hardware support. To this end, Bacon and Goldstein [19] originally proposed recording all snooping coherence transactions, which, while fast, produced a serial and voluminous log. Today, replay systems needs hardware and software components for recording and replaying execution.

This survey is divided into total of six sections. Section 1 consists of introduction and deterministic replay applications. In Section 2 we talk about hardware-based replay methods and Comparison them. In section 3 we bring some software-based deterministic replay methods and Comparison them. Section 4 consist a software-hardware schema. Section 5 gives Comparison of three recent methods simulation. And Section 6 concludes with the conclusion and our future work.

II. HARDWARE-BASED DETERMINISTIC REPLAY

Most hardware-based replay methods make major advances toward achieving parallel deterministic replay, but are limited [25]. The idea in these systems is to use special hardware to detect how the memory accesses of the different threads interleave during the execution and save such information in a log. Later, the log is used to replay, recreating the same memory access interleaving. The advantages of hardware-based replay methods is low overhead on run time and logging space, these methods also can cope with multiprocessor systems efficiently, they are largely impractical for use in realistic systems [25]. Disadvantages of hardware-based replay methods are that they focus only on the hardware implementation of the basic primitives for recording and, sometimes, replay. They do not address key issues such as how to separate software that is being recorded or replayed from software that should execute in a standard manner or from other software that should be recorded or replayed separately. This limitation is problematic because practical replay systems require much more than just efficient hardware for the basic operations [25].

A. Hardware-Based Approaches

Several hardware-based schemes have been proposed for deterministic multiprocessor replay.

Bacon and Goldstein [19] captured dependences between concurrent threads by logging the coherence messages in the bus of a multiprocessor using an attached board. This scheme is virtually free to probe effect: logging is done in hardware using extra bandwidth available on the bus. Because logging is centralized on a board with its own dedicated memory and optional disk, there is no effect on available resources for the CPUs.

The Flight Data Recorder (FDR) [7] is a full-system recorder for directory-based multiprocessors under SC. Like Bacon and Goldstein’s scheme [19], FDR observes coherence messages between processors. It improves on the former scheme notably by implementing a hardware version of Netzer’s Transitive Reduction (TR) optimization [8]. FDR approach for multiprocessor systems that continuously logs execution activity to enable full-system offline deterministic replay of a time interval preceding a software trigger. This method develop a complete solution that logs all system activities, including OS and I/O. Flight Data Recorder can be “always on” in anticipation of being triggered, because time and space overheads are modest. Also authors in this method, quantitative result for four commercial workloads and discuss how results scale with number of processors and memory size.

The BugNet [16] method focuses on recording only one application. BugNet presents a hardware solution for recording the inputs to the application. It does not focus on how to log the interleaving of the application threads. BugNet records user processes by storing the result of all load instructions within a hardware-based dictionary. Recorded information (less than 1MB) can be communicated back to the developer and used by them to characterize the bug by deterministically replaying the program’s execution before the crash. BugNet focuses on replaying only user code and shared libraries to find application level bugs. To achieve this, BugNet’s logs for a checkpoint interval contain the register state at the start of the interval and a trace of first load memory accesses. This is enough information to achieve deterministic replay of a program’s execution, without having to replay what goes on during interrupts and system calls. This results in small log sizes, where a log size of around 500KB is enough to capture a replay window size of 10 million instructions. This is small enough to motivate users to communicate the log back to the developer. The BugNet method has very little performance overhead, and the area overhead is around 48 KB for the few hardware buffers required.

The Strata [18] maintains a preprocessor counter that records the number of memory operations issued by a processor. Before a dependence-forming memory operation completes, Strata logs the memory operation counts of all the processors. This mechanism for shared memory dependency logging allows deterministic replay debugging for these multi-core systems. A stratum is logged across all of the processors every time a shared memory dependency needs to be captured. We log a stratum for every RAW and WAW dependency, and WAR dependencies are determined through offline analysis. A stratum provides a strict time ordering between memory operations that occurred before and after the stratum across the processors. The strata log is 5.8x smaller without compression and 12x smaller with compression than the log used in the previous point-to-point logging approach [15], [16]. Another advantage is that our strata approach requires less hardware than the point-to-point approach. In addition, based on the notion of strata, authors designed a shared memory dependency logging solution for snoop-based architectures, which the previous proposals did not address.

Xu et al. [8] extend FDR by introducing Regulated Transitive Reduction (RTR). This scheme introduces artificial dependencies so that Netzer’s TR can eliminate additional dependencies. This method is a new recording algorithm that significantly improves memory race recording in four aspects:
the log size, the hardware cost, the complexity and the applicability. The Regulated Transitive Reduction (RTR) regulates how memory races are replayed to reduce the log size. RTR’s key novelty is in creating stricter and vectorizable dependencies in the dependence log. The Set/LRU timestamp approximation method computes more accurate timestamps for uncached blocks and enables a significant reduction in both the log size and the hardware cost. Moreover, this method, decouple timestamp memory (D-TSM) to reduce the hardware complexity of the recorder. Also, this method authors extend race recording to handle TSO (x86-like), as well as SC to broaden its applicability. RTR, optimize race recording that can benefit many applications of deterministic replay and may encourage adoptions of hardware race recorders.

DeLorean [4] is a novel scheme for deterministic replay where processors execute groups of instructions atomically. Rather than recording data dependences, it records the total order of instructions executed. To support this operation efficiently, each processor in DeLorean executes instructions grouped into chunks, and only the total sequence of chunks is recorded. An arbiter serializes and records chunk commits. DeLorean has two fundamental advantages over current schemes. First, it records at the speed of the most aggressive memory consistency models used today—and also replays at high speed. This makes it useful for production-run debugging. Second, it summarizes the execution interleaving into a truly small log. DeLorean’s execution modes offer a trade-off between performance and log size. In OrderOnly, DeLorean records at the speed of RC execution and replays at 82% of RC speed. In contrast, most other schemes record only at the speed of SC execution and provide no details on replay speed. RTR presents an algorithm for recording TSO executions but does not evaluate its impact on execution speed or log size. Moreover, OrderOnly only needs 1.3 bits of compressed memory-ordering log per processor per kilo-instruction and, with stratification, only 0.6 bits. We estimate the latter to be 7.5% of the log size needed by Basic RTR. In PicoLog mode, DeLorean reduces the memory-ordering log to 0.05 bits per processor and kilo-instruction, which we estimate is 0.6% of the log size in Basic RTR. In this mode, we estimate that the total memory-ordering log of an 8-processor 5-GHz machine is only about 20GB per day. Recording speed decreases to 86% of RC execution speed (still higher than typical SC speed). Overall, we conclude that DeLorean greatly enhances the potential of deterministic replay to help debug multithreaded codes.

The Rerun [1] uses an intermediate approach. It also detects data dependences between processors but stores sequences of instruction groups in the logs. Rerun is a memory race recording mechanism that uses small hardware state (~166 bytes/core), writes a small race log (~4 bytes/kilo instruction), and operates well as the number of cores per system scales (e.g., to 16 cores). Rerun exploits the dual of conventional wisdom in race recording: Rather than record information about individual memory accesses that conflict, Rerun record how long a thread executes without conflicting with other threads. In particular, Rerun passively creates atomic episodes as the fundamental unit of ordering. Each episode is a dynamic instruction sequence that a thread happens to execute without interacting with other threads. Episodes are created passively by observing system behavior without altering the normal execution flow. When recording is enabled, the entire system execution is viewed as a collection of ordered episodes so that every dynamic instruction logically resides within the boundaries of an episode. Rerun records the outcome of an execution by simply logging the length and order of episodes. A causal ordering among episodes from different threads is established using Lamport Scalar Clocks [3], which is a standard mechanism from distributed systems used to create a global notion of logical time in systems where no single point of ordering exists. Using this mechanism, Rerun associates each episode with a timestamp that correctly places the episode in a partial order of execution and preserves inter-thread dependencies.

B. Comparison of Hardware-Base Approaches

Most of hardware-base schemes log and replay the full system execution, including application, OS and other threads [25]. The exception is BugNet [19], which focuses on recording only one application. BugNet presents a hardware solution for recording the inputs to the application. It does not focus on how to log the interleaving of the application threads.

Consider figure 1 [4] for comparing FDR [7], RTR [8] and Strata [18]. Figure 1(a) illustrates TR. Processor P1 writes locations a and b, and later P2 accesses b and a. The dependence 1:Wa→2:Ra does not need to be recorded because it is transitively implied by 1:Wa→1:WB, 1:WB→2:WB, and 2:WB→2:Ra. Consequently, FDR only records 1:WB→2:WB. FDR saves the processor ID and instruction count of the two instructions in a Memory Races Log buffer. BugNet [16] reuses FDR’s hardware to replay user code and shared libraries. It efficiently records the output of all load instructions by compressing them with a hardware-based dictionary scheme.

![Figure 1](image-url)  
Fig. 1. Key insights of previous work on deterministic replay: FDR (a), RTR (b), and Strata (c).

Figure 1(b) illustrates RTR. The code has dependences 1:Wa→2:Ra and 1:WB→2:WB. RTR introduces artificial dependence 1:WB→2:Ra, which is recorded. Now, RT eliminates the need to record the other two dependences. RTR also saves space by representing recurring dependences with a vector notation. The second contribution of Xu et al.’s work is a recording algorithm for a TSO machine [16]. Supporting TSO is significant because TSO is used in real machines. However, the authors do not evaluate the impact of the new algorithm on execution speed or on log size [16]. We refer to
Xu et al.'s work [16] as the RTR system, and distinguish between the Base (no TSO) and Advanced (TSO) support. The Base RTR support logs about 1B per processor per kilo instruction (compressed).

Figure 1(c) shows a reference trace with the points (S0 and S1) where strata are logged. Right before the second reference of the dependence 1:Wa→2:Ra is issued, Strata logs the memory reference counts of all 3 processors. The same process is repeated right before the second reference of the dependence 2:Wb→1:Wb. The other two dependences in the figure do not require the creation of a new stratum each of them already has its two references in different strata regions. Strata works with directory (and snoop-based systems) both under SC. Strata can choose to ignore WAR dependences when building the log. In this case, WAR dependences are uncovered at replay at the cost of slowing down the replay with multiple reexecutions [6]. The compressed log for 4 processors is 2.2KB per 1M memory references.

Both Wisconsin Rerun [1] and Illinois DeLorean [4] approaches seek to enable deterministic replay by focusing on recording how long threads execute without interacting. Rerun makes few changes to standard multicore hardware, while DeLorean promises much smaller log sizes and higher replay speeds. Future work includes improving Rerun's replay speed, generalizing DeLorean's hardware design alternatives, and making the original multithreaded executions more deterministic.

III. SOFTWARE-BASED DETERMINISTIC REPLAY

Software-based deterministic replay systems, uses software systems for recording all sources of non-determinism, such as network inputs or interrupts, and uses this information to guide the software down the same execution path during replay. Software-based schemes work well for uniprocessor systems; they have high run-time overhead on multiprocessor machines because current techniques for interposing on shared-memory accesses are inefficient. Fortunately, hardware-based schemes can record shared-memory interleaving efficiently [25].

A. Software-Based Approaches

There are several recent proposals for software-based deterministic replay.

Bressoud, et al [29] uses execution replay to enable a high-availability primary-backup system. The main system (or primary) is logged, and the logs fed to one or more backup systems, which replay the logs immediately. This guarantees that the backup system is in the same state as the logging system, ready to take over in the event of a failure. Since this system is targeted at single processor virtual machines, and does not address the issue of sharing.

Flashback [28] uses shadow processes to enable efficient rollback of the memory state of a process while a lightweight logger records the process interaction with the OS. Flashback records and replays processes by modifying the OS to log all sources of non-determinism during recording. This includes logging all results of system calls, plus any data the kernel copies into the process. For example, if a process makes a read system call, Flashback records the return value of the system call and the data that the kernel copies into the read buffer. When replaying, Flashback injects this same data back into the process when it encounters this specific system call. Flashback uses shadow processes to enable efficient rollback of the memory state of a process while a lightweight logger records the process interaction with the OS.

M. Russinovich, et al in [30] can replay the scheduling order between host processes. This strategy is difficult because a host process can be interrupted while executing in kernel mode (e.g. while executing a system call). It is hard to identify the point in the kernel where an interrupt occurred, yet identifying this point is critical for replaying the exact scheduling order. The hardware performance counters we used to identify the exact interrupt point in ReVirt do not work well when the interrupt point is in the kernel, because we configure them to count only user-mode instructions. Configuring them to count both user and kernel instructions also leads to difficulties—the kernel does not execute deterministically, so the instruction counts would differ during replay.

ReVirt [31] applies virtual-machine and fault-tolerance techniques to enable a system administrator to replay the long-term, instruction-by-instruction execution of a computer system. Because the target operating system and target applications run within a virtual machine, ReVirt can replay the execution before, during, and after the intruder compromises the system. This capability is especially useful for determining and fixing the damage the intruder inflicted after compromising the system. Because ReVirt logs all non-deterministic events, it can replay non-deterministic attacks and executions, such as those that trigger race conditions. Finally, because ReVirt can replay instruction-by-instruction sequences, it can provide arbitrarily detailed observations about what transpired on the system. ReVirt adds reasonable time and space overhead. The overhead for virtualization ranges from imperceptible for interactive and CPU-bound applications to 13–58% for kernel-intensive applications. The time overhead of logging ranges from 0-8%, and logging traffic for our workloads can be stored on a single disk for several months.

W. Dunlap, et al [32] presented their work on SMP-ReVirt, the first system to log and replay multiprocessor virtual machines on commodity hardware. They use hardware page protections to detect races between virtual CPUs in a multiprocessor virtual machine. This allows us to log and replay an entire virtual machine, including the kernel and all applications, without modifying the software. Using the hardware page protections avoids the overhead of instrumenting every read and writes in software, but necessitates a large granularity of sharing and incurs higher overhead when sharing occurs. Some workloads perform poorly, but others perform surprisingly well. The main factors that influence performance include how much the workload involves the guest kernel, and how sensitive the workload is to false sharing at larger sharing granularities.
B. Comparison of Software-Base Approaches

Software-based schemes work well for uniprocessor systems; they have high run time overhead on multiprocessor machines because current techniques for interposing on shared-memory accesses are inefficient. These systems make major strides toward achieving deterministic replaying, but fall short in one or more areas [25]. ReVirt [31] only works for uniprocessors and the extensions of this technique (SMP-ReVirt) for replaying multi-threaded applications on multiprocessors have substantial logging requirements and poor recording and replay speeds.

IV. Software-Hardware Interfaces Deterministic Replay

Software-Hardware schemas are combination of the best of the hardware-based and software-based replay schemes for multiprocessor systems. One best software-hardware schema have presented in [25] with name Capo. Capo can deterministic replay in multiprocessors efficiently with assistant of software and hardware.

While hardware-based replay proposals achieve low run time overhead, have low logging space overhead, and can cope with multiprocessor systems efficiently, they are largely impractical for use in realistic systems. The main problem is that recording and replaying in these methods are based on hardware assistant. They do not address key issues such as how to separate software that is being recorded or replayed from software that should execute in a standard manner (i.e., without being recorded or replayed), or from other software that should be recorded or replayed separately [25]. This limitation is problematic because practical replay systems require much more than just efficient hardware for the basic operations. Capo is a software-hardware interface for practical hardware-assisted deterministic replay of programs running on multiprocessor systems [25]. A key abstraction in Capo is the Replay Sphere, which allows system designers to separate cleanly the responsibilities of the hardware and the software components. Both the hardware and software components of Capo can be implemented in a variety of ways. In this discussion, for clarity, authors refer to a replay system where an OS provides the ability to record and replay processes or groups of processes. The same ideas also apply to any privileged software layer that records and replays unprivileged software running above (e.g., a VMM that records and replays VMs or groups of VMs).

To enable practical replay, Capo provides an abstraction for separating independent workloads that may be recording, replaying, or executing in a standard manner concurrently. For this abstraction to be useful, it must provide a clear separation between the responsibilities of software and hardware mechanisms. Moreover, it must also be meaningful to software components and yet still low-level enough to map efficiently to hardware. In Capo, the hardware records in a log the interleaving of the R-threads running within the same sphere. The software records the other sources of non-determinism that may affect the execution path of the R-threads, such as systemcall return values and signals.

Authors also design and build CapoOne, a prototype of a deterministic multiprocessor replay system that implements Capo using Linux and simulated DeLorean hardware. Our evaluation of 4-processor executions shows that CapoOne largely records with the efficiency of hardware-based schemes and the flexibility of software-based schemes.

V. Comparison of Three Recent Methods

In this section, we show result of three methods Rerun, Delorean and Capo simulation with SESC simulator [34] based on Log Size and Performance. In Delorean method, we consider OrderOnly mode.

A. Log Size

Depending on used method for deterministic replaying in parallel executions, the manufacture log size of executions can be variant. Figure 2 shows the size of the logs on three methods Rerun, Delorean and Capo in bits per processor per kilo-instruction.

![Fig. 2. Comparing log size (in bits per kilo-instruction) based on three standards Apache, jbb and SP2-G.M.](image)

B. Performance

Figure 3 compares the performance of three methods Rerun, Delorean and Capo under three standards Apache, jbb and SP2-G.M.

![Fig. 3. performance comparison.](image)

VI. Conclusions and Future Works

A system exhibiting deterministic replay capability can record sufficient information during an execution to enable a replayer to (later) create an equivalent execution despite inherent sources of nondeterminism that exist in modern computer systems. The information required includes initial state (e.g., a checkpoint) and nondeterministic events [7].

In this survey we introduced hardware-base approaches [19], [7], [16], [18], [8], [4] and [1] for deterministic replay on multiprocessors. They propose special hardware to detect the interleaving of memory accesses or instructions from the
different processors during execution, and save the key information in a log. This log is later used to replay the same interleaving. Also we talked about software-base approaches [29], [28], [30], [31] and [32]. They propose software systems that record all sources of non-determinism, such as network inputs or interrupts, and use this information to guide the software down the same execution path during replay. Cape is a software-hardware mechanism [25]. In Cape, the hardware records in a log the interleaving of the R-threads running within the same sphere. The software records the other sources of non-determinism that may affect the execution path of the R-threads, such as system call return values and signals.

We would like to research work explore several interesting topics in deterministic replaying. Our studies will improve a new approach for deterministic replay with small log size and better performance.

REFERENCES

