Probabilistic Replay of Java Programs on Multiprocessors

João M. Silva
joao.m.silva@ist.utl.pt

Instituto Superior Técnico
INESC-ID

Abstract. Alongside the rise of multiprocessor machines, the concurrent programming model has grown to near ubiquity, being the only possibility for developers wanting to take full advantage of their performance. Unfortunately, reasoning about concurrent programs is hard and bugs that stem from unanticipated interactions between tasks are easy to hatch. On the contrary, finding and fixing these bugs is a complex and time-consuming endeavour. To make matters worse, the debugging tools that programmers have relied on for decades fall short when applied to concurrent programs. Systems capable of replaying non-deterministic executions have long been proposed as a solution for this limitation of conventional methods.

We survey previous work on deterministic replay systems and present a taxonomy for classifying them. Moreover, we describe the objectives of our future work and how they are to be achieved by proposing an architecture and evaluation methodology for a replayer of concurrent Java programs. The key property of the system is its probabilistic approach: partial production run information is traced and an offline exploration of untraced non-deterministic space is used to achieve deterministic replay. This approach enables an efficient recording phase, at the cost of a potentially longer replay phase which, being done offline, is not as performance critical.

1 Introduction

Taking on the role of detective to debug complex systems is an all-too-familiar task for software developers. A usual debugging session starts with the programmer postulating a set of possible causes for the error. He then adds trace statements or breakpoints to the faulting system with the purpose of examining the state at a given point in its execution, hoping to make observations that allow him to confirm or reject each possible cause. Until the real cause is located, the programmer uses the gathered information to update the set of possible causes and iterates. This methodology is called cyclic debugging and has been the conventional way of debugging sequential programs for a long time [23].

1.1 The Challenge of Concurrent Programs

Cyclic debugging makes one assumption about the program: it must be deterministic in regards to the failure, i.e., it must always fail in the same way when given the same input. This is a fair assumption when dealing with sequential programs, because their only sources of non-determinism are input, which can usually be easily reproduced by the programmer, and certain external conditions such as time, which are rarely involved in bugs.

However, once we attempt to apply cyclic debugging to concurrent programs, the limitations brought on by this assumption become clear. Unlike their sequential counterparts, concurrent programs are inherently non-deterministic. Because they do not specify all possible execution paths, even if the program is fed the same input, successive executions may behave differently depending on the particular resolutions of race conditions among tasks. Furthermore, in many cases, concurrent bugs, such as atomicity violations and deadlocks, arise from very specific outcomes to race conditions, making them extremely rare. Thus, it is very frustrating and time-consuming for the programmer to iteratively zoom in on the cause of the error, mainly due to the time it takes to reproduce it. In fact, a recent study on real world concurrency bugs has shown that the time needed to fix such a bug is mainly taken up by the task of reproducing it [21]. Besides being time-consuming, any attempts to gain additional information about concurrency bugs may impose timing variations and contribute to the bugs' evasiveness, a phenomenon called the probe effect.

1.2 Deterministic Replay

Most solutions that tackle the problem of debugging concurrent programs attempt to move their non-determinism out of the way, allowing for the employment of traditional techniques, such as cyclic debugging. Deterministic replay has long been suggested as a means to achieve this goal, typically supported by re-execution approaches. Such systems typically operate in two phases: a record phase, in which the outcomes of non-deterministic events occurring in an execution of the target system are traced; and a replay phase, in which the replayer forces another execution of the same target to experience the same outcomes to non-deterministic events as the recorded execution. A fitting metaphor for deterministic replay is that of a time travelling machine [16], because it grants a debugger the ability of travelling backwards in an execution to inspect past states of the program.

We distinguish two types of non-determinism, with which these systems must deal with, in order to achieve fully faithful replay: input non-determinism, which arises from events such as system calls, I/O operations, signals and interruptions; and memory non-determinism caused by the interleaving of accesses to shared memory.

Input non-determinism is present on sequential and concurrent programs on both uniprocessor and multiprocessor machines. Several software-based systems have been developed that efficiently replay this kind of non-determinism, including Flashback [40], Jockey [35] and others [4, 41, 8, 16, 10, 44, 9, 12, 25, 1, 29].

Memory non-determinism is caused by memory races, which can be further separated into synchronization and data races. Synchronization races are intentional and beneficial, as they allow for competition among tasks to enter a critical section, for example. Their absence would turn deterministic any program running on uniprocessors and data race free programs on multiprocessors. Data races, on the other hand, are mostly unintended and the result of faulty synchronization.

Recording the task scheduler's decisions is enough to reproduce synchronization races. Systems that employ this technique can be made efficient in software and work well on uniprocessors [20, 33, 11]. However, the parallelism between tasks on a single processor machine is simulated because, at a given point in real time, there is only one task executing and accessing memory. In contrast, multi-processor machines allow for truly concurrent execution of tasks, making the replay of scheduling decisions no longer sufficient. Somehow, the deterministic replay system must now record the outcome or the ordering of every memory access to any location that is potentially shared. There is extensive work done on this problem. Software-only systems are possible but suffer from significant space and time overheads and result in a lot of probe effect, making them unsuitable for online recording [5, 9]. Thus, hardware and hybrid systems have been created, benefiting from low runtime overhead and little to no probe effect, at the cost of being impractical for use in most realistic scenarios [2, 42, 26, 43, 24, 25]. Very recently, work on probabilistic replay systems has shown potential of enabling the creation of software recorders which are usable during production runs [1, 29].

The rise of multi-core and multi-processor machines has made concurrency the obligatory model for any system that requires high performance. Unfortunately, the pervasiveness of this concurrent programming model is the culprit behind the easy to hatch and hard to detect concurrency bugs. There is a need for easily deployable execution replay solutions that are efficient enough to use during production runs without sacrificing too much of the program's performance.

1.3 Document Road-map

The rest of the document is organized as follows: Section 2 states the objectives of our work; Section 3 surveys and classifies related work on the topic of deterministic replay; Section 4 proposes an architecture for our solution; Section 5 describes the methodology we will use to evaluate the developed system; and Section 6 summarizes this document.

2 Objectives

The goal of this work is to extend a Java virtual execution environment to incorporate a deterministic replay mechanism. More specifically, the replay system should have the following properties.

- **Low record overhead.** Time overhead should be low enough to enable the recorder to be active during production runs of the program without severely crippling its performance. Similarly, space overhead should be low as to enable long recording periods.

- **Faithful user-level replay.** The system should record an execution at the abstraction level of an application. The resulting replay execution should be faithful to the original execution in regards to specified properties, such as the occurrence of a bug.
- **Multiprocessor and data race support.** The system ought to replay executions of applications running in multiprocessor machines. Moreover, this property should not compromise the ability to reproduce the outcomes of data races. Indeed, given that data races are at the root of many concurrency bugs, we feel that placing a constraint on target programs that forces them to be perfectly synchronized is very limiting and even somewhat unreasonable.
- **Easy and inexpensive deployment.** The system should operate on unmodified binaries, precluding the need to have access to source code, which may be hard to modify, highly variable or even unavailable/proprietary. Preferably, it should rely on modifications to the Java virtual machine and/or binary instrumentation at load-time.

3 Related Work

Our work builds mainly upon the research area of deterministic replay. Other topics, such as concurrent debugging or race detection algorithms, are also somewhat related, but contribute little, in practice, to the design of the system we propose to create. Therefore, we will do an in-depth survey and analysis of deterministic replay systems and propose a new taxonomy that, we believe, offers improvement and better understanding of the topic than previous classification efforts do.

3.1 Deterministic Replay

Deterministic replay is ambiguous to some extent, so we further specify that we analyse systems that enable deterministic replay through re-execution of a particular non-deterministic execution. For instance, deterministic replay achieved by making concurrent programs inherently deterministic are not considered.

To our knowledge, there is a lack of up-to-date surveys with sufficient depth on this topic. In fact, the surveys that do exist seem limited both in number of systems considered and/or number of criteria used to categorize them [31, 15, 6, 32, 7]. Most divide systems based mainly on the type of algorithm (data- vs. order-based) or on the implementation substrate (software vs. hardware) and only consider a few extra criteria. Thus, one of the contributions of our ongoing work is a new taxonomy to enable a better understanding of the subject of deterministic replay.

3.2 Usage Models

Deterministic replay systems have been developed and deployed to enable a wide range of applications, including the following:

- **Debugging.** The vast majority of deterministic replay systems have been developed with the purpose of allowing programmers to employ usual debugging techniques for deterministic systems, like cyclic debugging, on non-deterministic ones [20, 2, 28, 34, 5, 33, 17, 41, 42, 11, 40, 27, 35, 16, 26, 10, 24, 12, 1, 29, 14]. Some attempt to facilitate debugging by providing mechanisms that provide the illusion of reverse execution [16].
- **Fault tolerance.** Deterministic replay can be used as an efficient means for a fault-tolerant system to maintain replicas and recover after experiencing a fault [4].
- **Security.** Systems have been built that use deterministic replay to find exploits for vulnerabilities, to run security checks [31] and to examine the way attacks were carried out [8]. They enable these by allowing the system administrator to inspect the state of the system before, after and during an attack.
- **Trace collection.** Trace collection can be made efficient and inexpensive by using deterministic replay technology to compress large execution traces [44].
- **General replay.** Some systems have been developed as general replayers, with the intent of enabling more than one of the prior usage models within a single solution [9, 25, 43, 13].

3.3 Abstraction Level

One of the most important design decisions to make when creating a deterministic replay system is the level of abstraction at which the execution of the target system will be recorded and replayed. The choice defines not only the scope and power of the replayer, but also the specific sources of non-determinism it will have to face, which we discuss in detail in Section 3.4. Furthermore, the choice will place constraints on the techniques one can use to implement the replay system.

In practice, the abstraction level appears to be either the user or the system levels of the software stack. Among the replay systems that operate at user-level, some replay only application code [34, 41, 40, 10, 12, 1, 29, 20, 28, 5, 33, 17, 11, 14], while others replay shared library code as well [27, 35, 25]. System-level replayers enable reproduction of executions of whole systems, including OS code [2, 42, 26, 43, 9, 24, 13, 4, 8, 16, 44].

The decision between implementing a software-only replayer or a hardware-assisted one is highly conditioned by the abstraction level at which it is to operate. Indeed, if one wants to replay at system-level, the options are either a hardware- or a virtual machine-based replayer. User-level replayers, on the other hand, can and are mostly implemented completely in software.

3.4 Types of Non-determinism

To achieve faithful deterministic replay, one has to record every single source of non-determinism that might cause deviation between two executions of the same program. The sources of non-determinism can be divided into two sets: (1) input non-determinism, which amounts to any kind of input that a program receives from external sources; and (2) memory non-determinism, which arises from the interleaving of different threads and the resulting interleaving of shared memory accesses.

The techniques used in recording and replaying these two different sources of non-determinism are very distinct. In fact, from our survey of existing replay systems, we found that the type of non-determinism a system deals with is the most distinguishing criterion when attempting to characterize it. For instance, most properties of memory non-determinism replay systems are not applicable to input non-determinism replay systems. Therefore, in our framework of deterministic replay, systems are separated into these two categories: (1) systems that replay input non-determinism; and (2) systems that replay memory non-determinism.

Of course, some systems do deal with both sources of non-determinism, in which case their two facets will be discussed separately. Because the techniques used to handle the two kinds of non-determinism are quite distinct, they enable themselves to be discussed independently from one another. Thus, this approach to the discussion of the systems does not hinder their analysis or understanding.

3.4.1 Input Non-determinism

Input non-determinism occurs in both sequential and concurrent executions. It can arise from any input to a layer that is not generated by that same layer. Moreover, input events may be non-deterministic with respect to both their data and timing. For instance, a system call is only non-deterministic in relation to the data it returns or manipulates, since its timing can be derived from program order. On the other hand, interrupt and DMA operations are additionally non-deterministic with respect to timing, due to their asynchronicity.

The actual instances of non-determinism are dependent on the level of abstraction with which we consider the target system. Most deterministic replay systems record and replay at either the user [34, 41, 40, 10, 12, 1, 29, 27, 35, 25] or the system level [4, 8, 42, 16, 44, 9, 24], having to deal with very distinct input.

User level. At user-level, most input is generated by the OS. Deterministic replay systems must handle the following sources of non-determinism [31]:

- *System calls.* There are quite a few system calls that are non-deterministic. A prominent and easily understandable example is the UNIX system call `gettimeofday`, which is dependent on timing-related external conditions. System calls that read from disk or a network card are also non-deterministic, because the data present in these devices may change between executions. The data read from networks is particularly difficult to reproduce manually, when compared with data from the disk. Even memory allocation system calls like UNIX's `malloc` are non-deterministic, because their return value is dependent on the current internal state of the OS.

- *API calls.* Sometimes, applications do not invoke system calls directly. Instead, they invoke an high-level API. Java programs, for example, access the system through the Java API. Native programs may also access the system through a library, such as libc. Thus, one may replay these programs by recording at the API level instead of the system call level.
- *Signals.* The OS delivers asynchronous signals to applications in order to notify them of a variety of events. The occurrence and timing of such a signal makes the control flow of an execution non-deterministic.
- *Non-deterministic user-level architectural instructions.* The Instruction Set Architecture (ISA) of a processor may contain non-deterministic instructions available in user-mode. The `rdtsc` x86 instruction is one such instruction, since it reads the CPU's timestamp/cycle counter.
- *Stack and dynamic library locations.* Even though not technically an input to the program, the locations in memory of both the program stack and dynamically loaded libraries may cause non-determinism across program executions.

System level. At the system level, input is generated by the hardware itself. The following sources of non-determinism must be handled [31]:

- *I/O.* Any information read by the system from an I/O device is potentially non-deterministic. As examples, the data in a hard drive may be rewritten and the data provided by a network card is clearly timing dependent. When communication with these devices is done through memory mapped I/O, any data read from the assigned addresses must be recorded and reintroduced during replay.
- *Interrupts.* Hardware interrupts cause the processor to stop whatever it is doing, save its context and branch to a routine that handles the particular interrupt being raised. This effectively modifies the control flow of the execution. Furthermore, interrupts are asynchronous, meaning the point at which the processor stops cannot be predicted. Thus, both the timing and the contents of each interrupt have to be recorded and reintroduced at the same point upon replay. Note that, in contrast, traps do not have to receive this treatment, because they are raised by the processor itself as a result of a faulty condition when executing an instruction. This means that both the timing and contents of a trap are dependent solely on the particular instruction and its operands. If the replay is successful up to the execution of the instruction, the operands should be the same during replay as during recording and an equal trap will be naturally raised by the processor.
- *Direct Memory Access (DMA).* Direct memory accesses allow devices to write directly to memory, bypassing the processor. Therefore, in the processor's point of view, they are asynchronous events. Just like interrupts, both their timing and written values must be recorded.
- *Non-deterministic architectural instructions.* The results of executing any non-deterministic instruction featured in the processor's ISA also have to be recorded.

3.4.2 Memory Non-determinism

In contrast with input non-determinism, memory non-determinism is unique to concurrent executions. The phenomenon behind this kind of non-determinism is called a memory race, which occurs whenever (1) there are two unsynchronized accesses to the same shared memory location and (2) at least one of those accesses is a write operation. It is necessary to further distinguish between two types of memory races: synchronization races, which occur between synchronization operations; and data races, which occur between data accesses.

Synchronization races are intended and beneficial. They allow for competition between threads to access a critical region or lock a mutex, for example. Removing synchronization races would turn a concurrent execution into a sequential one. Nevertheless, the outcome of these races must be recorded. In fact, recording and replaying synchronization races is enough to replay any concurrent program running on a uniprocessor system. This is the case because, in such systems, the parallelism between threads is just an abstraction, since only one thread may execute and access memory at a given point in real time. Thus, the outcome of memory races is dictated solely by the order in which tasks are scheduled to execute.

Data races, on the contrary, are the reason behind many concurrent bugs. They usually arise from a faulty or non-existent synchronization between accesses to shared objects. Data races make the job of a deterministic replay system a lot harder on multiprocessor machines, because when multiple processors exist, parallelism is no longer a simple matter of abstraction, but a real physical phenomenon. Therefore, the outcome of data races stops being solely dependent on scheduling decisions, i.e., knowing which tasks are executing on each processor at a given point in time is not enough to know which one will win a particular data race. We are now face-to-face with a situation in which any pair of accesses to shared memory is a potential data race, the outcome of which must be recorded. This is the major problem that deterministic replay systems face today, because recording the order of every access or the data read by them incurs too much time and space overhead, especially for software-only solutions.

Note that the distinction between synchronization and data races is merely a matter of abstraction. Synchronization races are actually caused by data races on a certain synchronization object, such as a spin lock. However, such objects amount to only a tiny fraction of the whole memory space, while the more general data races can occur on any object. Thus, any replay system capable of reproducing the outcome of data races can also reproduce the outcome of synchronization races.

3.5 Replay Start Point

Before delving into the record and replay techniques that enable deterministic replay of input and memory non-determinism, let us consider the starting point of a replay. The obvious place to start is at the beginning of the program's execution. Given that the events of interest occur fairly early in the execution, this approach is fine. However, consider the following scenario: a bug manifested itself during an execution of a program after it had been running for three days. In such a case, even if the replay system is efficient enough to allow recording during the whole execution, a programmer would take three days to complete an iteration of cyclic debugging. This problem encouraged replay system designers to develop and/or deploy checkpointing techniques to allow a replay to start at arbitrary points of an execution. In a way, one can think of a checkpoint as a compressed log, allowing the replay system to fast-forward through parts of the execution that are known to contain no events of interest. Additionally, checkpoints are needed to provide the illusion of reverse execution in an efficient way.

We now survey the techniques used in deterministic replay systems to create checkpoints and enable multiple replay starting points.

Flashback [40] is a user-level replayer that uses shadow processes to create checkpoints and efficiently roll back the state of a target process. The user (or automated debugger) can request a checkpoint at any point during the execution. Flashback then creates a snapshot of the process, stores it as a shadow process structure in the kernel, and immediately suspends the resulting shadow process. The user can then go back in time to the point when the shadow process was created and Flashback is able to replay from that point forward. The overhead of restoring checkpoints is reduced by restoring pages using a copy-on-write policy.

liblog [10] is a user-level, library-based replay system with a checkpointing mechanism based on libckpt [30]. This library writes allocated memory regions in a checkpoint file. A bootstrap application reads this file and overwrites its own memory to conform with the contents of the file, effectively becoming equivalent to the program at the point the checkpoint was created.

Jockey [35] is another user-level, library-based replayer. It creates checkpoints using a technique based on Flashback and libckpt. The target process is forked and the child creates the checkpoint file, while the original process continues running.

FDR [42] is a hardware-based replayer that enables deterministic replay of a whole system during the last second before a fault occurred. Due to its somewhat unique goal, checkpointing is mandatory and must be performed often. FDR was tested with the SafeyNet [39] checkpointing mechanism. The checkpoint itself contains the architectural state of all processors, an image of physical memory and I/O state. Because the image of physical memory is large, FDR incrementally creates logical checkpoints by saving the values of memory locations that are overwritten. A checkpoint can then be recovered from the system's final state by undoing changes made to memory. Due to the nature of

FDR, checkpoints can be discarded when a new one is created. Thus, even though checkpointing is employed, FDR cannot start replaying from multiple points in time, only the most recent.

BugNet [27] is another hardware-based replayer, but it records at user-level. It uses the notion of a checkpoint interval to achieve replay of concurrent programs. At the beginning of an interval, the architectural state (program counter and register values) is saved. From then on, in contrast to what was done by FDR, it saves only the value at a certain memory location the first time it is accessed after the interval began. This amounts to a reduction in both log size and hardware cost.

TTVM [16] is a virtual machine-based, system-level replayer. Its checkpoints comprise a complete state of the virtual machine: CPU registers, physical memory and virtual disk, among others. To improve efficiency, a copy-on-write policy is used on both memory and disk, to save only the pages that have been modified since the last checkpoint. These saved updates to memory and disk can be used as undo or redo logs to time-travel between checkpoints.

3.6 Replaying Input Non-determinism

Sequential systems suffer as much from input non-determinism as their concurrent counterparts. How is it, then, that programmers have gotten away with debugging them using techniques like cyclic programming for so long? The reason is that, for most programs, this kind of input can be reproduced with relative ease: files can be restored, network activity can be created manually and signals or interrupts are rarely a problem. Deterministic replay of input non-determinism becomes relevant only when we assume that the programmer cannot re-create the input or when its reproduction is a resource heavy task.

To ensure deterministic replay in regards to input non-determinism, the replay system must make sure the target system perceives no difference in its interaction with external resources during re-execution. The solution is to log the inputs listed in Section 3.4.1 during a recording phase and injecting them back upon replay.

The following sections survey real systems that have the goal of replaying input non-determinism at user-level and system-level. Table 1 summarizes the surveyed systems according to the criteria that compose our taxonomy for input non-determinism replayers.

3.6.1 User-level Replay

At this abstraction level, the major sources of non-determinism are system calls and signals. We focus on techniques that enable their replay. Non-deterministic instructions are a somewhat lesser problem.

Flashback [40] records system call level input using kernel modifications. More specifically, system calls are hijacked by replacing the default handler for each one with a wrapper function that handles the logging and replaying. The results and side-effects of each system call are logged when recording and injected back in during replay. System calls that affect the application's state only, such as `getimeofday` or `getpid`, are the easiest to replay. They need not be re-executed by the OS, meaning the call can be bypassed and the program's state is simply modified according to the logged results. Other system calls change the state of the OS itself and need to be re-executed during replay in a way that ensures the same state modifications for the application that had occurred during the record phase. The syscalls `malloc` and `fork` are examples of this latter case. Creating the wrappers for all system calls is a tedious and far from general solution, as each recorded routine must be paid special attention.

Flashback does not handle signals. Nevertheless, the authors propose using the approach described by Slye and Elnozahy [37] to achieve deterministic signal reproduction. In this approach, a signal is annotated with the increment suffered by the instruction counter, available in most architectures, since the last asynchronous event occurred. This would uniquely identify the timing of the signal.

Capo [25] is a hybrid software-hardware system that aims at replaying application and shared library code. It can reproduce both input and memory non-determinism, with the former being a responsibility of the software part of the system. CapoOne, its prototype implementation, takes advantage of small kernel modifications and uses the Linux `ptrace` process tracing mechanism to control the target processes. The mechanism for dealing with system calls is equal to the one used in Flashback.

It improves upon Flashback in that it also replays signals, but the mechanism for capturing their exact timing is not specified in their paper.

ODR [1] records system calls and non-deterministic instructions by having a signal delivered to itself (in the context of the process being executed) by a small set of kernel modifications whenever the target system performs such actions. System calls are replayed by around 200 manually written stubs, but, unlike Flashback, these are executed when handling the signals delivered by the modified OS, not by substituting the default system call routines.

Jockey [35] differs from Flashback and Capo in that a runtime user-mode library is injected into the target application to enable deterministic replay. Jockey logs a mix of system calls, libc calls and non-deterministic instructions by replacing them with calls to stubs that handle the recording and replaying.

The correctness of Jockey can be broken, due to the way the timing of signals is recorded. These are associated with the closest successive stub call, instead of a point in time. During replay, the delivery of the signal to the application is, thus, delayed until that stub call completes. In practice this approach may have a very high probability of reproducing a concurrent bug, but it breaks the replayer's full correctness nonetheless.

liblog [10] is another library-based replay system. It uses very similar techniques to the ones employed by Jockey to record system calls, libc calls and signals. It does not seem to record non-deterministic instructions.

jRapture [41] is a replay tool for Java programs. It operates by recording at the Java API level, as most interactions between a Java application and the system are done through this interface. It is implemented as a set of modified versions of the Java API classes. Just like system calls, methods in these classes can have side-effects that span further than their return value. As a result, each modified class must be written by hand. jRapture can also replay native methods that are called through the Java Native Interface (JNI) at the cost of becoming platform dependent.

R2 [12] provides a different approach to library-based replay systems. Jockey, liblog and jRapture all log statically defined interfaces: system calls and libc calls for the two former and the Java API for the latter. In contrast, R2 enables recording and replaying of a user-defined interface. Anything above the interface is re-executed during replay, while anything below is bypassed and the results read from a log. Choosing the right interface is a trade-off between the amount of information that is logged and the detail of the replay. The higher the level is, the lesser detail the replay has, since a big chunk of the program is not re-executed and, thus, is impossible to analyse. The consequence could be the reproduction of bug symptoms without actually reproducing the bug.

Since the interface is user-defined, the stubs for each chosen function must be created on-the-fly. *R2* provides the user with an annotation language through which the side-effects of a function can be made explicit, enabling the creation of the stubs.

PRES [29] is a very recent system focused on replaying concurrent programs on multiprocessors. It replays system calls and signals, at least. The uncertainty about non-deterministic instructions stems from the fact that, due to space limitations on their paper, the handling of input non-determinism is downplayed to only a few lines, while the rest of the space is used to describe how they deal with memory non-determinism. This is indicative of the fact that replaying input non-determinism is considered a problem that has been solved efficiently by previous research.

3.6.2 System-level Replay

Replaying at system-level imposes more constraints on the implementation options of deterministic replay systems. It is not possible to record all system-level events using a software-based solution that runs in user mode. Thus, it comes as no surprise that systems replaying at this abstraction level are implemented as either hardware modifications or inside virtual machines. The input that needs to be recorded includes non-deterministic instructions, I/O, interrupts and DMA operations.

Bressoud & Schneider [4] pioneered the idea of using virtual machine technology to achieve deterministic replay of whole systems. They use execution replay to enable a high-availability primary-backup system in which the primary machine is recorded and the backup systems use the recording to follow the primary. With this setup, the backups accompany the state changes of the primary with absolute faithfulness and are ready to take over in the event of failure. Space overhead is not a problem, because when all backups have replayed a certain part of the execution, the corresponding log portion can be discarded.

To record the timing of asynchronous events like interrupts or DMA operations, their system uses the *recovery register* of the HP's PA-RISC architecture. This register is decremented whenever an instruction is executed and an interrupt is delivered to the Hypervisor when it becomes negative. This behaviour allows the system to regain control at a very specific point in the execution and deliver a virtual interrupt to the backup, for example.

ReVirt [8] also takes advantage of a virtual machine, but with the goal of allowing the system administrator to inspect the execution during an attack to the system. Input from external devices, non-deterministic system calls to the host OS and non-deterministic instructions are logged. The point at which a virtual interrupt is delivered to the guest is uniquely identified by combining the program counter with the hardware retired branches counter. The instruction counter points to a specific instruction in the system's code, but not a specific execution of that instruction, since subsequent branching instructions may lead to it being executed multiple times. The retired branches counter does what the name implies, it increments whenever a branch instruction executes. Thus, the combination of the program counter with such a branch counter uniquely identifies a specific point in an execution.

TTVM [16] and ***SMP-ReVirt*** [9] are two more virtual machine-based replay systems used to debug operating systems and general replay, respectively. They handle input non-determinism with the same techniques that ReVirt uses.

ReTrace [44] was developed by *VMWare* to reduce the overhead of collecting arbitrarily complex traces of production executions using deterministic replay. It records all input to which a virtual machine is subjected. Asynchronous events are associated with a point in time by keeping track of the number of instructions executed.

FDR [42] is a hardware-based replayer. It records I/O by storing load values and interrupts by using an instruction counter to uniquely identify their timing. As for DMA writes, FDR models each DMA interface as a pseudo-processor and uses the same algorithm that handles memory races (discussed in Section 3.7.5). This is possible because DMA operations use the same directory protocol to maintain cache coherence as processors. The recorded information is kept in hardware buffers.

DeLorean [24] is another hardware-assisted approach to deterministic replay. It uses a shared DMA log and two per-processor logs for I/O and interrupts. Like FDR, it models DMA interfaces as pseudo-processors and makes them go through the same chunk commit protocol that processors use to record memory non-determinism (details in Section 3.7.5). Interrupt timing is identified by the `chunkID` of the chunk that initiates execution of its interrupt handler.

3.6.3 Software vs Hardware Approaches

Hardware approaches like *FDR* and *DeLorean* achieve the lowest performance overhead. For instance, when recording an execution of the Apache web server, *FDR* has a performance overhead below 2%, including memory non-determinism logging. In spite of this superiority, software approaches have been shown to enable deterministic replay of input non-determinism quite efficiently, achieving performance overheads below 10% during the record phase [16, 40, 44, 8].

Comparing approaches in regards to space overhead is not straightforward. Replay systems differ in events logged, compression schemes and use different benchmarks in their evaluation. Nonetheless, *Flashback* claims its log size grows linearly with the number of system calls the target program issues. We would expect most other systems' logs to grow in a similar fashion: linearly with the number of events they log.

The slightly better recording performance does not seem to justify the cost involved in employing hardware support. This conclusion is supported by the fact that only two of the surveyed systems

Table 1. Overview of input non-determinism deterministic replay systems.

	Bressoud & Schneider [4]	jRapture [41]	ReVirt [8]	FDR [42]	Flashback [40]	Jockey [35]	TTVM [16]	liblog [10]	ReTrace [44]	SMP-ReVirt [9]	DeLorean [24]	R2 [12]	Capo [25]	ODR [1]	PRES [29]
Abstraction Level															
System	×		×	×			×		×	×	×				
User + Library						×							×		
User		×			×			×				×		×	×
Type of Inputs															
System Calls					×	×		×					×	×	×
API Calls		×				×		×				×			
Signals						×		×					×	×	×
Non-deterministic Instructions	×		×			×	×		×	×				×	
I/O	×		×	×			×		×	×	×				
Interrupts	×		×	×			×		×	×	×				
DMA	×		×	×			×		×	×	×				
Start Point															
Static	×	×	×	×								×	×	×	
Dynamic					×	×	×	×	×	×	×				×
Implementation															
Hardware				×							×				
Software															
Library-based		×				×		×				×			
Binary Instrumentation															×
OS Modifications					×								×	×	
VM Modifications	×		×				×		×	×					
Usage Model															
Debugging		×		×	×	×	×	×			×	×		×	×
Fault-Tolerance	×														
Security			×												
Trace Collection									×						
General Replay										×			×		

that handle input non-determinism are implemented in hardware. Furthermore, both support input replay for completeness purposes, as their reason for being implemented in hardware is memory non-determinism replay.

3.7 Replaying Memory Non-determinism

Until now we have only looked into solutions for replaying sequential programs or concurrent programs in which no dependence exists between tasks. These are only subjected to input non-determinism, which can be efficiently replayed with both software and hardware approaches. When tasks communicate with each other through whatever means, they become dependent on each other. Systems that behave in this manner exhibit memory non-determinism, which adds a lot of complexity to the process of recording and replaying their executions.

In this section we will first present the criteria we use to classify a system capable of memory non-determinism deterministic replay, in regards to (1) the model of its target system, (2) the recording

mechanism and (3) the replaying mechanism. We then survey real replay systems that handle memory non-determinism. Table 2 summarizes the characteristics of each system by classifying them according to our taxonomy.

3.7.1 Target System Model

Multiprocessor Support. There is a very significant difference between a concurrent system execution on a uniprocessor and on a multiprocessor. In the presence of a single processor, parallelism is only an abstraction, since there can be no point in real time in which multiple tasks are actually being executed. With more than one processor, the opposite is true: tasks may execute concurrently in a real world sense. Thus, while in a uniprocessor the interleaving of tasks is only dependent on the points at which a task is swapped by another, in a multiprocessor tasks can interleave in very complex ways. Indeed, since the processors offer no exact guarantees about the time taken to execute a portion of code, knowing when each task was swapped in is not enough to derive the interleaving.

Due to the increased complexity involved in replaying systems executing on multiple processors, some replay systems focus only on uniprocessors [34, 5, 10]. Some replayers designed for uniprocessors can actually replay multiprocessor executions (e.g. DejaVu [5]). However, they impose a total order on synchronization and memory operations that would effectively simulate a uniprocessor during replay. In our classification we do not consider these systems as supporting multiprocessor replay. Despite the involved complexity, most surveyed systems support replay on multiple processor machines [20, 2, 28, 33, 42, 11, 27, 26, 43, 9, 24, 13, 25, 1, 29, 14].

Data-race Support. Data races occur when two concurrent tasks issue unsynchronized memory operations and at least one of them is a write. A synchronization race is a special case of data race, in which the non-determinism is intended, because it enables competition between tasks. Recording only synchronization races is far simpler and efficient then recording data races. Unsurprisingly, systems that focus on uniprocessors, where scheduler decisions fully define the interleaving of tasks, never explicitly record data races. However, their outcome is implied by the outcome of synchronization races. In fact, they are supported by all replayers that only support single processor executions [34, 5, 10].

In the light of these facts, data races need only be recorded explicitly when the replay system tackles executions on multiprocessors. Our survey shows that most replayers which support multiprocessor executions also provide some form of support for data race replay [2, 28, 42, 27, 26, 43, 9, 24, 13, 25, 1, 29, 14]. Some, on the other hand, still try to avoid the hassle of recording data races [20, 33, 11] by placing a constraint on the target system: it must be perfectly synchronized. Nonetheless, these replay systems can faithfully replay an execution up to the point at which the first data race occurs. In addition, they may be coupled with a data-race detector to enable debugging of imperfectly synchronized target systems [33].

Task Creation Model. A replay system's mechanisms may place constraints on how tasks are created in the target system. We consider two models for task creation: (a) a static model in which the number of tasks is fixed and known a priori, and (b) a dynamic model in which tasks can be created and destroyed freely throughout the recorded execution.

We found out that most replayers do not force the target system to conform to a static model of task creation. All replay systems operating at the user level of abstraction support dynamic creation of tasks. Furthermore, since all system-level replayers are based on recording the behaviour of processors instead of individual user-level processes or threads, they inherently allow for dynamic task creation. They mostly do, however, assume a static number of processors, but this is a very reasonable assumption. Only Capo [25] takes measures to enable a variable amount of processors, but only between the record phase and the replay phase.

From all the surveyed systems, only Netzer's Transitive Reduction algorithm [28], which was never implemented by its authors, forces the target system to conform to the static task creation model. This is due to its use of vector clocks, which contain a fixed number of positions, one for each task [18, 22]. The algorithm could, however, be subject to extension by using dynamic vector clocks to handle task creation and termination, at the cost of larger space and time overheads [19]. An implementation of Netzer's TR was developed for the system-level, hardware-based replayer FDR [42], but it was modified to use scalar instead of vector clocks [18, 22].

3.7.2 Recording Mechanism

Algorithm type. The algorithms used to record memory non-determinism can be classified as either content-based or order-based.

- *Content-based.* The most straightforward approach is a pure content-based algorithm: a recorder simply stores the data read by each instruction in a log file and the same data is then fed back in during replay. Such an approach would clearly generate logs of enormous size, making it far too inefficient for any practical use. Despite this apparent impracticability, some deterministic replay systems have managed to follow a content-based approach and achieve reasonable efficiency by recording only a subset of the data read by instructions [27].
  There is a benefit which is unique to this sort of recording algorithm: one is allowed to replay individual or subsets of tasks. Since all memory input is recorded, the tasks that generated it are obsolete. It may, however, be argued that this benefit is somewhat useless, because replaying tasks in isolation makes it harder to analyse the complex interactions they may have had with others during the recording phase [20].

- *Order-based.* Instead of attempting to restore the state seen by each instruction, order-based approaches take advantage of the fact that memory contains most of the state of a system. It is observed that there is no need to record all data read by instructions, but only external data that is written to memory. In concurrent systems, tasks influence the memory state of systems and, depending on the timing of write operations, may alter the behaviour of one another. Thus, in an order-based approach, the recorder stores the relative order of critical events, such as shared-memory accesses or synchronization operations. Then, the runtime environment is set up in an initial state equivalent to the one of the recorded execution. Finally, the system is executed and, whenever a situation arises in which the execution could deviate from the original, the log is used to nudge it in the right direction. In other words, the tasks are forced to access shared memory in the same order as in the original execution, forcing the data in memory to undergo the same chain of read and write operations, as well as the corresponding sequence of values.
  The main advantage of this approach is that most of the data read by instructions is reproduced by the system and, thus, needs not be recorded. This can also be seen as a shortcoming, since instructions can no longer be executed in isolation. Nonetheless, a total order between all critical events generally takes up lesser trace bandwidth than recording all the data read from memory. A reflection of this fact is that all but one of the surveyed memory non-determinism replay systems use an order-based algorithm [20, 2, 28, 34, 5, 33, 42, 11, 26, 10, 43, 9, 24, 13, 25, 1, 29, 14].

No matter what approach we use, it is clear that, in their purest forms, both are too inefficient for practical purposes. They record at too low an abstraction level, resulting in a lot of trace data. Only by raising the level of abstraction and designing techniques to record subsets of trace data can a deterministic replay system become feasible.

As a final note, let us remark that the order-based approach is only available when replaying memory non-determinism. A replayer for input non-determinism cannot guarantee that external sources inject the right input into the target system, at the right time. The only solution is, therefore, to store the contents of those inputs.

Traced events. Recording memory non-determinism can be achieved by tracing different kinds of events, such as:

- *Shared-memory Accesses.* Unsynchronized shared-memory accesses are behind every instance of memory non-determinism, though the latter only occurs if one of the accesses is a write operation. Systems that directly trace accesses to shared memory generally support both data races and multiprocessor executions.

- *Synchronization Operations.* Another way of recording memory non-determinism is to simply trace the partial or total order of synchronization operations, such as those that manipulate mutexes or monitors. This type of recording only provides enough information to reproduce synchronization races. As a consequence, no system that records solely the order of synchronization operations is able to simultaneously support multiprocessor executions and data races [33, 11].

- *Task Schedule.* As previously mentioned, reproducing the task schedule of the target system is enough to replay its executions, given that they occur on a single processor machine [34, 10]. A replay system that traces only task schedule can also be applied to multiprocessor executions, but the target system must be perfectly synchronized. Therefore, this trace method is as limited as tracing synchronization operations.
- *Chunk Commits.* Recent hardware replay systems use a chunk-based approach, in which the order of chunk commits is traced [24, 25, 13]. A chunk represents a block of instructions that are executed without conflicting with each other in terms of memory accesses. Since a chunk cannot commit unless it does not conflict with another, data races are logged implicitly.

Sharing Identification. Replay systems that trace shared-memory accesses need a way to identify them among the local memory accesses. If this cannot be achieved, every memory access is considered as potentially shared, which would lead to an impracticable amount of trace information. There are three ways of detecting shared access events used in the surveyed systems:

- *High-level Constructs.* If all accesses to shared-memory are done through well-defined high-level constructs at either language or OS level, the work of the recorder is very simplified [20, 5]. This approach makes the replay system dependent on a particular protocol for accessing shared objects.
- *Dynamic.* Most recorders detect accesses to shared-memory dynamically, as the system executes. There are a lot of techniques that enable this approach, from using scalar or vector clocks [33, 11, 5, 28] to spying on cache coherence messages [2, 42, 26, 13] and using hardware page protections [27], among others.
- *Static.* Finally, one surveyed system [14] employs static analysis on the target system prior to execution to identify a conservative set of shared objects, i.e., a superset of the set of shared objects.

Trace Optimization. Many of the recorders we surveyed apply optimization techniques to the events they trace in order to achieve a better log size.

A very common optimization for systems that record shared-memory accesses is transitive reduction, which identifies redundant constraints on the ordering of accesses and removes them from the log. Netzer [28] proposed an algorithm to find the optimal set of constraints that are enough to imply all possible constraints, which has been implemented in FDR [42]. An improvement over this algorithm, called Regulated Transitive Reduction (RTR) [43], introduces artificial constraints to further reduce the set that needs to be explicitly stored. Systems that use transitive reduction optimization include FDR [42], PRES [29] and SMP-ReVirt [9].

Another frequent optimization is to record intervals instead of individual events. As an example, if a recorder logs shared-memory accesses, and multiple successive accesses are done by the same task, these may be represented as an interval. DejaVu [5], Bacon & Goldstein [2] and LEAP [14] employ this kind of trace file compression.

A third optimization is to represent timestamps as increments, instead of their whole value. Furthermore, we can store only non-deterministic increments by taking a certain increment (e.g. +1) to be the deterministic one. RecPlay [33] and JaRec [11] use this optimization.

Finally, data-based approaches can take advantage of the fact that in the absence of external entities, the target system can regenerate the values read by most instructions without the help of the replay system. BugNet [27] uses checkpoint intervals to identify parts of an execution that meet this criterion and records only the value read by the first load operation of the checkpoint interval to each memory location.

3.7.3 Replay Mechanism

Determinism. All memory non-determinism has its roots on data races. No matter whether all of them are recorded or how the recording is accomplished, replayers may provide different levels of replay fidelity. Even though research on record and replay technology has largely had high fidelity replay as an objective, some recent propositions attempt to relax this guarantee in order to reduce recording overhead.

- *Value Determinism.* Under value determinism, the replayed execution reads and writes the same values to and from memory, at the same execution points, as the recorded execution [1]. This kind of determinism has been used from the very first deterministic replay systems all the way to very recent work [20, 2, 28, 34, 5, 33, 42, 11, 27, 26, 10, 43, 9, 24, 13, 25, 14]. It provides a high fidelity replay of the original execution.
- *Conditional Determinism.* A different approach was taken by the designers of PRES [29], who decided to relax the fidelity guarantees by providing replay executions that match partial trace of the original execution and user-defined conditions. We call this approach conditional determinism. This guarantee relaxation allows the replay system not to record the outcome of every data race during recording. This also means that the replay execution can differ substantially from the original. However, the faulting condition is replayed, so the outcomes of data races used during replay are still significant for fixing the problem.
- *Output Determinism.* The authors of ODR [1] proposed a replay system that provided output determinism, i.e., the replayed execution outputs the same values as the original run (an output is defined as any value sent to devices). Output determinism offers weaker fidelity guarantees than value or conditional determinism, since it does not enforce non-output properties of the original execution, such as the values read from memory. As a consequence, only output-visible failures such as assertion violations, crashes, core dumps and corrupted data can be reproduced. Failures that produce no distinctive output, such as a deadlock, cannot be reproduced.
  It is argued that, despite its limitation, output determinism is enough for debugging purposes for two reasons: (1) it can reproduce all output-visible failures, and (2) it provides memory-access values that are consistent with the failure, even if they are distinct from those which originally caused the failure.
  Finally, the key benefit of output determinism is the same of conditional determinism: since the outcome of data races can differ from the original, they need not be fully recorded.

Probabilistic Replay. Debugging a concurrent program without replay support usually involves executing the system a lot of times until the bug is finally reproduced. Most replay systems, on the other hand, reduce the number of attempts to just one, but at the cost of a possibly high recording overhead. A very recent idea in deterministic replay is to explore the space between these two approaches to debugging [29, 1]. In other words, the replay system may need a few attempts (e.g. 5-10) to replay the bug, but provides the benefit of reduced recording overhead by only partially tracing the original execution. Since the recorder provides only a partial trace of the original execution, there must be a mechanism that, during the replay phase, is able to somehow reconstruct an equivalent execution to the original in regards to some fidelity guarantee.

Both PRES [29] and ODR [1] can record partial traces of the original execution at different levels: from recording only synchronization operations to tracing all shared-memory accesses. Then, at the beginning of the replay phase, the space of possible executions that fit the partial trace is intelligently explored until the fault manifests itself. The executions performed during this process are fully traced, which enables 100% successful replays after the fault is reproduced for the first time.

A final problem remains unsolved: how do these systems know when the bug has occurred? The answer lies with the relaxed fidelity guarantees that were previously discussed. PRES and ODR offer only conditional and output determinism, respectively. PRES analyses the state of the replayed execution when visible events occur to check whether the conditions provided by the user are true [29], while ODR finds the execution that produces a certain output by using a formula solver [1].

3.7.4 Software-only Solutions

It is no surprise that software-based deterministic replay systems mostly operate at user-level. The exception to the rule is SMP-ReVirt [9] which is implemented as a virtual machine to record at system-level. Those that record at user-level are implemented either through user libraries [20, 33, 10], OS modifications [34] or binary instrumentation, which is performed either statically [14] or dynamically, using a process-level VM [1, 29] or a high-level language VM [5, 11]. It is also noteworthy that none of the software-only systems use data-based algorithms.

Synchronization Race Approaches. Systems that record scheduling decisions or synchronization operations reproduce only synchronization races, in order to avoid the overhead that recording all

data races incurs. As a result, they are unable to simultaneously replay multiprocessor executions and to support imperfectly synchronized programs.

Russinovich & Cogswell proposed a replay system with the purpose of reproducing uniprocessor executions of concurrent programs [34]. The target program is instrumented at compile time to maintain a software instruction counter. A modified Mach OS supplies the replay system with the precise points at which context switches occur. They report overheads around 10-15% during the record phase and slightly higher during the replay phase, which are reasonable for a production run. Being implemented as a set of OS modifications makes the solution highly dependent on that particular OS.

liblog [10] is a library-based replay system that, besides recording input non-determinism and enabling replay of distributed applications, records the thread schedule of each process, which is enough to replay a multithreaded process on a uniprocessor. It faces the interesting challenge of recording the schedule using only a user-level library. Even though this is an easy task for OS- or VM-based replayers, a user-level library does not often enjoy the privileges of observing context switches, much less controlling them. The solution was to impose a user-level cooperative scheduler on top of the OS scheduler. Since it records `libc` calls, liblog only makes context switches when such a call is issued and also delivers signals at those points, which sacrifices fidelity for a convenient way to keep the library in control of the execution during replay.

Instant Replay [20] assumes that the program manipulates shared objects through coarse-grain synchronization operations that implement a CREW (Concurrent Read Exclusive Write) protocol. Because only these operations are reproduced, Instant Replay does not support data races. To generate a total order of object accesses, the synchronization operations are instrumented to increment a version number, associated with each shared object, like a scalar Lamport clock. Instant Replay traces the version number upon every read operation and the number of read operations between writes. Since no compression method is used to reduce the trace file, it can become very large. During replay, read operations wait until the version number is correct and write operations wait until the correct number of reads has been performed. Depending on how coarse-grained the synchronization operations are, the performance overhead of the system may differ greatly — fine-grained synchronization will incur high overhead.

DejaVu [5] (and ***Distributed DejaVu*** [17]) records the logical thread schedule of Java programs, consisting of the ordering of synchronization operations (`monitorenter`, `monitorexit`, `wait`, `notify`, `suspend`/`resume` and `interrupt`) and shared-memory accesses. However, since shared objects are identified through the use of the Java `volatile` keyword, they are always accessed in a safe way and can be modelled as a synchronization operation. The recording mechanism uses scalar Lamport clocks: a global, and a local for each thread. The process yields a total order, making it impossible to replay multiprocessor executions without simulating a uniprocessor. The trace file itself is optimized by recording intervals instead of whole sets of timestamps. Implementation-wise, the system makes modifications to the JVM's synchronization routines. The authors succeed in producing small trace files and report recording overheads below 100%.

RecPlay [33] is another system that traces synchronization operations. However, it uses scalar Lamport clocks, not to create a total order, but a partial order, enabling parallelism during replay of multiprocessor executions. Their ROLT (reconstruction of Lamport timestamps) method produces a trace, for each thread, consisting of a sequence of timestamps. The trace is compressed by storing only non-deterministic increments instead of whole timestamps. Since only synchronization races are recorded, replays are only guaranteed to be correct up to the first data race. RecPlay attempts to minimize this inconvenience by employing data race detection during the first replay execution, using a method based on vector clocks. Performance-wise, the authors report an average of about 20% slowdown during the record phase, making it quite efficient.

JaRec [11] is Java record/replay system implemented using bytecode instrumentation. Loaded classes are passed to an instrumentor through the JVMPI (JVM Profiler Interface). No modifications to the JVM are needed, making the approach completely portable. JaRec traces all synchronization operations available in Java, including synchronized methods and blocks (monitors), `wait` and `notify`

calls, and the `start` and `join` methods of threads. A scalar Lamport clock is associated with each thread and synchronization object to create a partial order between synchronization operations. The partial order enables parallelism during replay and, thus, replaying of multiprocessor executions. Nonetheless, imperfectly synchronized programs are not supported past the first data race. The generated trace file is similar to that of RecPlay and is compressed using the same techniques.

Data Race Approaches. Since replaying solely synchronization races prevents the replay system from simultaneously supporting multiprocessor executions and imperfectly synchronized programs, there has been substantial work done with the goal of reproducing all data races. Doing so is generally a high overhead process which makes it difficult for software-only systems to be efficient enough to be enabled during production runs. This fact led to many hardware-assisted solutions which will be surveyed in Section 3.7.5. For now, we survey the software solutions that tackle the problem.

Netzer introduced an algorithm for tracing the optimal set of ordering constraints necessary to reproduce an execution, named Transitive Reduction (TR) [28]. The key feature of the algorithm is that the trace file can be reduced by removing dependencies between shared-memory accesses that are implied by other dependencies. For example, if `T1:1` → `T2:4` and `T1:2` → `T2:3` are dependencies detected between tasks `T1` and `T2`, then the first dependence need not be stored. Note that `T2` waits at instruction `3` for `T1` to reach instruction `2`, which implies that upon reaching instruction `4`, `T1` will have already executed instruction `1`.

To track dependences, the algorithm uses vector clocks attached to each process and shared memory location. Every time a process accesses a shared-memory location, both clocks are compared and updated, which has the potential for introducing a lot of overhead. Another disadvantage of the algorithm is that vector clocks must have a slot for each task, forcing the target system to follow a static task creation model. This shortcoming can be overcome by employing dynamic vector clocks to handle task creation and termination [19]. The authors did not implement the algorithm, so no practical results about its performance are presented.

SMP-ReVirt [9] is the only software-based approach that replays at system-level. It is implemented using a virtual machine and employs a unique solution for detecting shared-memory accesses through the use of hardware page protections. More specifically, processors have different privileges for each memory page and, when a processor attempts to access a page to which it holds no access privileges, SMP-ReVirt increases them while lowering those of another processor. The log is made up of the points at which privileges change. This mechanism has two substantial limitations.

Firstly, the granularity of sharing is limited to the size of a page, leading the system to succumb to false sharing when fine-grained sharing is used and as the number of processors increases. As a consequence, runtime overhead and trace file sizes scale very poorly. It is reported that the overhead can go up to 10x on machines with a modest number of processors.

Secondly, because SMP-ReVirt can only record the points at which the privileges of processors change, trace file optimizations such as Netzer's TR cannot be applied.

LEAP [14] is a replayer for Java programs that produces a partial order by tracing the threads that access each shared variable. As a result, it can reproduce multiprocessor executions. Synchronization operations are also traced and the trace file is optimized using intervals to represent successive accesses by the same thread. LEAP is the only surveyed system that employs a static technique to identify shared variables, instead of having them manually identified through some high-level construct or identifying them dynamically as the program executes. The technique is inherently conservative, which guarantees reproduction of every possible data race. The system is implemented by instrumenting the bytecode of the target program to generate a record and a replay version. The authors report results showing LEAP is about 10x faster than global order systems, 5x faster than Instant Replay and about 2x faster than Lamport clock-based approaches.

ODR [1] introduced the concept of output determinism, already discussed in Section 3.7.3. The authors argue that, for debugging purposes, the fidelity of value determinism is helpful, but unnecessary. By lowering its fidelity guarantees, ODR manages to free itself from the burden of recording the outcome of data races.

The trace of an execution consists of three sets: (1) the *input-trace*, which is the result of input non-determinism recording; (2) the *lock-order*, which is a total ordering of lock operations; and (3) the

path-sample, a set of tuples (t, c, l) where t is the thread, c is the instruction count and l is the program location of the instruction. The detail of each trace may vary and the missing pieces compose the state of possible executions. A depth-search algorithm explores that space using a formula solver to find executions that generate the same output as the original.

The combination of the lower fidelity guarantees and the offline state exploration stage yields a replayer that supports multiprocessor executions and imperfectly synchronized programs with low recording overhead, in exchange for a potentially costly replay.

PRES [29] is a probabilistic approach to replaying multiprocessor executions using a software-only recorder. Like ODR, it overcomes the overhead of recording data races by not tracing them (at least fully) in the original execution. Therefore, in order to reconstruct the partially recorded execution, an intelligent offline replayer searches the space of possible outcomes to non-recorded data races. The resulting executions are always consistent with the partial trace of the original, but the user must provide information that enables detection of a "correct" replay. We classify this guarantee as conditional determinism, because the replayed execution may differ from the original as long as the final state satisfies the user-provided conditions. The space of possible replays can be very large, but a feedback system is proposed that evaluates each failed replay and generates additional information to guide the next attempt. This mechanism results in most bugs being successfully replayed after a very modest number of replay attempts.

The authors also explore the space of possible methods for sketching the original execution. Starting from a baseline recorder that traced only input, signals and thread scheduling, they experiment with different recorders that incrementally store more information: global order of synchronization operations, system calls, functions, basic blocks and shared-memory operations. As the amount of traced events increases so does the overhead of recording, the speed of replaying and the faithfulness of the replay.

It is reported that PRES, using synchronization or system call global order tracing, significantly lowers the recording overhead of previous approaches. These results come at little cost, as most bugs were still reproduced in under 10 replay attempts. Furthermore, PRES scales well as the number of processors increases.

3.7.5 Hardware-assisted Solutions

The software-only solutions to memory non-determinism replay struggle a lot with the overhead involved in recording data races and also with the resulting probe effect. As a result, some systems sacrifice flexibility by replaying only uniprocessor executions or assuming data race freedom, while others relax fidelity guarantees and increase replay speed by deriving the outcomes of data races offline. Mainly as an answer to the limitations of software approaches, many researchers have worked on deterministic replay systems that take advantage of hardware support [2, 42, 27, 26, 43, 24, 13, 25]. Such support has the notable side effect of enabling replay of whole systems, unlike the user abstraction level of most software solutions. Nonetheless, a few hardware-assisted systems do record at the level of user libraries.

Point-to-point Approaches. Replay systems that track dependencies at the level of individual memory accesses are said to use a point-to-point approach. A timestamp is associated with each memory block and updated on every memory access.

Bacon & Goldstein were the first to propose replaying executions by spying on the cache coherence protocol of directory-based multiprocessors using hardware modifications [2]. They piggyback a hardware instruction counter on coherence messages to identify sharing. A subset of the messages is logged to generate a partial order of memory accesses. The replayer has little time overhead, but can generate huge logs.

FDR [42] can replay the last moments of the execution of a whole target system running on a directory-based multiprocessor. It augments cache blocks to contain a scalar clock and modifies the cache coherence protocol to carry and update them. By spying on the protocol's messages, FDR is able to derive the dependencies between memory accesses. It improves upon Bacon & Goldstein's approach substantially by implementing a modified version of Netzer's TR algorithm, in hardware, to

compress the trace of memory dependencies. Their version uses scalar clocks instead of vector clocks, which reduces the overhead in exchange for a slightly larger trace size.

The authors report that on a 4-processor server with commercial workloads, and given less than 7% of physical memory, FDR can record the last second of execution with less than 2% slowdown.

RTR [43] is an extension of FDR in which Netzer's TR algorithm is improved, resulting in the Regulated Transitive Reduction algorithm. Moreover, FDR's assumption of Sequential Consistency (SC) is relaxed to Total Order Store (TSO). RTR improves on TR by creating artificial dependencies that allow for further trace reduction. TSO is supported by having a hardware component that detects violations of SC and stores the loaded value instead of the usual ordering constraint. The overhead imposed during recording is as negligible as FDR's, but no evaluation of the benefits of RTR compression over Netzer's TR was made.

BugNet [27] supports deterministic replay of user code and shared libraries. The operation of the system revolves around checkpoint intervals, which start with the creation of a new checkpoint consisting only of register state. The value returned by memory load operations that first access a certain memory location in a particular interval are logged, while the values of following loads in that interval are derived by the program itself. A dictionary of common values is used to compress the trace. Given these mechanics, checkpoint intervals represent a set of committed instructions. Each interval has a maximum size and can be prematurely terminated by interrupts and context switches. To aid in debugging, BugNet also uses FDR's point-to-point approach to record shared memory dependencies, but this trace is unnecessary for replay.

Due to its data-based recording algorithm and checkpointing mechanism, BugNet can replay individual tasks and start the replay at the beginning of arbitrary checkpoint intervals.

Chunk-based Approaches. A chunk represents a block of instructions that are executed without conflicting with each other in terms of memory accesses. Enforcing the order of chunk commits is sufficient to replay the original execution. Chunk-based approaches can benefit from transitive reduction techniques just like point-to-point approaches.

Strata [26] proposes using a logging primitive called stratum. The system maintains an instruction counter for each processor in the machine and, whenever a conflicting memory access is to be performed, a vector with the current counter values for all processors is traced. This is analogous to committing a chunk that started when the previous stratum was recorded, but chunks are named strata regions. Stratums are only recorded if the first memory access in the conflict occurred in the previous strata region. This fact enables a transitive reduction algorithm more efficient than Netzer's TR, because a single stratum can capture multiple dependencies. The trace is reduced even further by not recording WAR (write after read) dependencies, because an offline analysis stage is able to derive the total order between memory accesses without them. Another advantage of Strata is that, unlike point-to-point approaches, the replayer is applicable in both snoop- and directory-based systems.

DeLorean [24] forces processors to execute instructions in chunks, which are invisible to software. When a chunk finishes executing, it requests a central module, the Arbiter, whether it can commit. Each chunk is associated with a signature based on Bloom Filters that is used by the Arbiter to immediately make the decision by comparing it with the signatures of already committed chunks.

The system can record in three modes: (1) *Order&Size* mode, in which both the size of chunks and their order is non-deterministic; (2) *OrderOnly* mode, in which chunking is deterministic; and (3) *PicoLog* mode, in which everything is deterministic. *PicoLog* mode requires no recording whatsoever, because the Arbiter forces a predefined chunk schedule (e.g. processors round-robin) and size during both the original and replay executions. In *OrderOnly* mode, DeLorean simply traces the order of chunk commits. Finally, in *Order&Size* mode, a log of chunk sizes is also maintained, because chunks can be truncated due to somewhat rare events. The purpose of these three modes of operation is to explore the trade-off between overhead and log size — the latter decreases while the former increases as we move from *Order&Size* to *OrderOnly* and then *PicoLog*.

Capo [25] is a software-hardware hybrid approach that operates at user level (including shared libraries). The key abstraction of the system is the notion of *Replay Sphere* that allows for the separation of duties between software and hardware modules, and enables multiple jobs running

in parallel (recording, replaying and standard execution). Each sphere is a group of threads that are recorded and replayed as a whole. Threads belonging to the same process must be part of the same sphere, but the latter may include threads from multiple processes. Tracking threads instead of processors provides more flexibility.

Memory non-determinism is handled by the hardware components of the system, given the overhead that it imposes when done in software. Capo places little constraints on the way the interleaving of memory accesses is recorded, which allows for integration with any of the hardware replayers discussed here. A prototype of Capo was built which used the DeLorean replay mechanism. Despite the additional abstractions taking a toll on both trace sizes (15% and 38% for engineering and system application, respectively) and recording overhead (21% and 41%) when compared with the original DeLorean, they are still modest. Concurrently recording two applications increased the overhead by 6% and 40% for the same classes of applications.

ReRun [13] records how long a thread executes without conflicting with another. The system passively creates atomic episodes analogous to chunks. Lamport clocks are used to establish and trace the interleaving of episodes. Their size is also recorded, as an episode must be terminated when it conflicts with another in terms of memory accesses. The detection of conflicts itself is done by piggybacking on the cache coherence protocol. Enforcing the size and interleaving of episodes is enough to replay the original execution.

The main advantage of ReRun is enabling scalable trace sizes on par with other hardware recorders, while requiring only a fraction of the hardware state. While FDR requires augmentations to all cache blocks, ReRun only needs a very small amount of state per processor.

3.8 Distributed Replay

We say a deterministic replay system is distributed if it is capable of cooperating with other instances of itself when replaying distributed programs. There are three major cases to consider: (1) the closed world case, in which all tasks involved in a distributed system are operating under the replayer's supervision, (2) the open world case, in which only one task is supervised by the replayer, and (3) the mixed world case, in which some tasks are supervised and others are not.

While the open world case can and has to be handled by recording network input to the tasks, note that such a replayer does not fit our definition, because it does not cooperate with others. Instead, the tasks are replayed individually by simulating the environment. Nonetheless, this mechanism has been the norm for deterministically replaying distributed systems.

In closed world situations, much space overhead can be avoided by having multiple replayer instances coordinate and regenerate network messages, instead of simulating them. ReVirt [8] proposes this optimization, but does not implement it. Distributed DejaVu [17] handles closed world cases by extending DejaVu's notion of critical event to encompass relevant network-related Java API calls and their paper is very explicit on how to handle stream-based communication, datagram-based communication and connections. liblog [10] uses Lamport clocks to replay communicating peers consistently.

Mixed environments can be handled as closed world cases for cooperating peers and open world cases for the rest. However, we must either know which peers are cooperative a priori, or have a discovery protocol in place that can find them without interfering with the communication protocols used by the distributed system.

3.9 Summary

Support for input or memory non-determinism is the most distinguishing criterion of deterministic replay, because the systems on both sides use very distinct techniques. Input non-determinism can be replayed efficiently with software-only solutions. On the other hand, memory non-determinism is only fully handled in an efficient way be hardware-assisted approaches. Software-based approaches struggle with recording data races and many avoid them altogether, recording solely synchronization races. Thus, they are unable to simultaneously support multiprocessor executions and imperfectly synchronized programs. However, recent probabilistic software-only approaches have shown potential for enabling efficient recording while supporting multiprocessors and imperfectly synchronized programs, at the cost of higher replay overhead.

Table 2. Overview of memory non-determinism deterministic replay systems.

	Instant Replay [20]	Bacon & Goldstein [2]	Netzer's TR [28]	Russinovich & Cogswell [34]	DejaVu [5]	RecPlay [33]	FDR [42]	JaRec [11]	BugNet [27]	Strata [26]	liblog [10]	RTR [43]	SMP-ReVirt [9]	DeLorean [24]	ReRun [13]	Capo [25]	ODR [1]	PRES [29]	LEAP [14]
Target System Model																			
Multiprocessor Support	×	×	×			×	×	×	×	×		×	×	×	×	×	×	×	×
Data Race Support		×	×	×	×		×		×	×	×	×	×	×	×	×	×	×	×
Dynamic Task Creation	×	×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Abstraction Level																			
System		×	–				×			×		×	×	×	×				
User + Library			–						×							×			
User	×		–	×	×	×		×			×						×	×	×
Record Mechanism																			
Traced Events																			
Shared-memory Accesses		×	×		×		×		×	×		×	×				×	×	×
Synchronization Ops.	×				×	×		×									×	×	×
Schedule				×							×								
Conflict-free Intervals														×	×	×			
Algorithm Type																			
Data-based									×										
Order-based	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×
Sharing Identification																			
High-level Constructs	×			–	×	–		–			–								
Dynamic		×	×	–		–	×	–	×	×	–	×	×	×	×	×	×	×	
Static				–		–		–			–								×
Trace Optimization		×	×		×	×	×	×	×	×		×	×		×			×	×
Replay Mechanism																			
Determinism																			
Value	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			×
Output																	×		
Conditional																		×	
Optimistic/Probabilistic																	×	×	
Dynamic Start Point									×		×	×	×	×				×	
Implementation																			
Hardware		×	–				×		×	×		×		×	×	×			
Software																			
Library-based	×		–			×					×								
Binary Instrumentation			–					×									×	×	×
OS Modifications			–	×	×														
VM Modifications			–										×						
Usage Model																			
Debugging	×	×	×	×	×	×	×	×	×	×	×			×			×	×	×
General Replay												×	×		×	×			

4 Architecture

Before presenting a concrete architecture for the system to be developed, we consider it important to state a few of high-level design decisions that stem directly from the intersection of our objectives with the related work.

Firstly, since our focus is on deterministic replay of Java user-level applications, a hardware-assisted approach would be severely misappropriate. Most Java applications are executed on software-

only virtual machines, which makes them the obvious platform for supporting the replay system. Specifically, we intend to modify the Jikes Research Virtual Machine, embracing both the perks and limitations of software-only deterministic replay.

Secondly, given a software-only platform, the goals of low recording overhead and data race support on multiprocessors severely limit our options. Indeed, only systems that relax fidelity guarantees and rely on partial recording followed by an offline intelligent search of non-deterministic space seem to provide these properties simultaneously [1, 29]. Thus, we decide to follow a probabilistic approach, complete with conditional determinism, i.e., a replay execution shall be guaranteed to be consistent with the partial trace of the original and to conform to a set of user-specified conditions about its state.

Finally, we will not tackle input non-determinism and checkpointing. Since many software solutions have been developed that efficiently solve these problems, we feel the effort of developing such solutions for our replayer would be misplaced. For completeness, existing solutions might be adopted, including in-house [36].

The remainder of this section proposes an architecture and process for a probabilistic Java deterministic replay system, illustrated in Figure 1. It takes the two design decisions just presented into account and draws a lot of inspiration from the architecture of PRES [29].

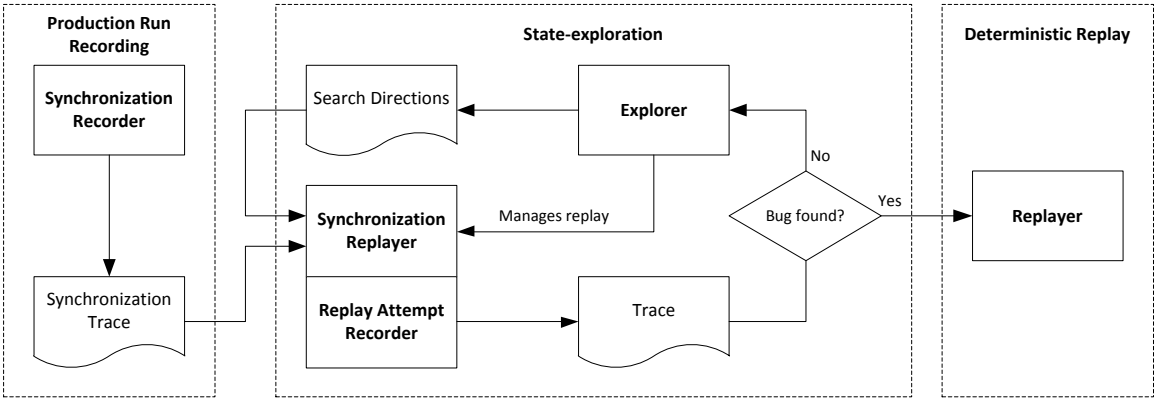


Fig. 1. Proposed architecture and process for a probabilistic Java deterministic replay system.

4.1 Process Phases

Traditional deterministic replay systems operate in two phases: (1) the record phase, in which a production run of the target program is traced, and (2) the replay phase, in which the execution traced in the first phase is reproduced deterministically. However, a probabilistic replayer does not trace enough information during the record phase to perform deterministic replay right away. It may take a considerable amount of attempts until an appropriate replay execution is performed. As such, even though the replayer would surely reduce the time for bug reproduction, debugging could still prove to be a frustrating task if this two phase model was to be used. To solve this problem, we divide the replay phase into (1) a state-exploration phase, in which a search for the bug is performed, and (2) a deterministic replay phase, in which, using new trace information collected during the search, the bug is reproduced in every attempt [29]. The result is a system operating in three distinct phases.

1. **Production Run Recording.** The production run of the target program is recorded by tracing only synchronization operations or scheduling decisions. This is the most time-critical phase, because it is performed online.

2. **State-exploration.** An explorer module runs and manages replay attempts. These are performed by a replayer that takes into account the synchronization trace produced in the previous phase and search directions provided by the explorer, consisting of extra fixed outcomes for non-deterministic events. All replay attempts are fully recorded (including data races) and, if a replay is unsuccessful, the resulting trace is used by the explorer to generate new search directions. Otherwise, the

trace contains all the data necessary for deterministically reproducing the bug. This phase of the process is potentially costly, but since it is performed offline, we argue that the costs are not unreasonable, given the benefits experienced during production run recording. Moreover, the search may be speeded up by leveraging checkpoints and by performing parallel exploration using the very multiprocessor machines the target itself runs on.

3. **Deterministic Replay.** In this final phase, a replayer uses the complete execution trace produced in the previous phase to deterministically replay the faulty condition with 100% probability.

4.2 Modules

We now discuss in more detail the modules of the system.

Synchronization Recorder. The module responsible for tracing the synchronization operations and/or scheduling decisions that occur during a production run of the target program, generating the *Synchronization Trace* file. We plan on employing a Lamport clock-based technique, similar to those proposed by RecPlay [33] and JaRec [11], to enable the recording. Optimization efforts should be directed towards this recorder, since it is the most performance critical component of the whole system, given that it operates online. Moreover, the less overhead imposed by the module, the less manifestation of the probe effect occurs.

Synchronization Replayer. An execution replayer that uses two sources of information to make decisions at non-deterministic points. The first is the *Synchronization Trace* file generated by the *Synchronization Recorder*, providing fixed outcomes for synchronization races. The second source is the *Search Directions* file provided by the *Explorer* module, containing constraints on the outcomes of specific data races and possibly a checkpoint representing the start point of the replay. These constraints represent the direction of the search and are created from the analysis of previous replay attempts. The replayer conforms to all specified synchronization race outcomes and data race outcome constraints, but it does not enforce any particular outcome for other non-deterministic events.

Replay Attempt Recorder. Every replay attempt is fully traced by this module. The resulting *Trace* file must contain fixed outcomes for all memory non-deterministic events that occurred in the execution. To facilitate development, we plan on using a Lamport clock-based technique like that proposed in DejaVu [5]. However, if time permits, a similar technique to the used used by LEAP [14] would be preferable, since DejaVu imposes a total order that allows for no concurrency during replay. The requirements of the *Replay Attempt Recorder* module make it a lot heavier in terms of performance than the *Synchronization Recorder*. This arises as a reasonable limitation, because the recorder operates offline.

Explorer. The module responsible for conducting the exploration on untraced non-deterministic space. The first function it performs is the management of replay attempts, from starting them to stopping them when the user-specified conditions are met or when the execution deviates from the original due to different outcomes of untraced non-deterministic events. The *Explorer* may also run multiple replay attempts simultaneously to take advantage of the multiprocessor machines where the target executes. The second function of the module is to create or update the *Search Directions* file with a set of constraints on the outcome of non-deterministic events after each replay attempt. The file may also contain a checkpoint that compresses part of the replay execution that precedes the non-deterministic event to be explored next and that has been executed in previous attempts.

Replayer. The module that deterministically replays the target program with 100% probability of bug reproduction. It follows the trace file resulting from the execution chosen by the *Explorer* during state-exploration, which contains outcomes for all memory non-deterministic events.

5 Evaluation

The evaluation will focus on performance (overhead imposed during the different phases) and replay correctness (the percentage of bugs successfully reproduced). It is to be performed by using the developed system to record and replay a micro-benchmark and real-world concurrency bugs in complex

Java applications. The micro-benchmark has the purpose of comparing the performance of our approach with that of state-of-the-art solutions, while real-world concurrency bugs provide unbiased validation of the usefulness of the system for practical uses.

5.1 Micro-benchmarking

The micro-benchmark will be designed to facilitate the comparison of execution overheads between our system and state-of-the-art replay solutions. The latter include: (a) DejaVu, a total-order-based recorder [5]; (b) JaRec, a partial-order-based recorder [11]; and (c) LEAP, a very recent approach capable of reproducing imperfectly synchronized executions on multiprocessors [14]. From these, only LEAP is publicly available and capable of handling the same use cases as our system. We intend to implement the other solutions and extend them where necessary, staying faithful to their corresponding publications.

The comparison will be focused on production-run recording, since it is the most critical phase of a deterministic replayer's operation. Furthermore, our system's replay phase differs too much from that of the other approaches, rendering any comparisons uninformative. Thus, we will use the two following metrics to evaluate the performance of each system:

- Production-run recording time overhead relative to bare execution of the program, measured as the percentage increment in execution time and average CPU load increase;
- Production-run recording space overhead, both in memory and disk, measured in number of recording points and trace file size increase per time unit.

Finally, the benchmark should be capable of exploring these measures across two dimensions: the number of threads, and the number of shared memory accesses and synchronization operations.

5.2 Real-world Applications

To perform an unbiased evaluation of our system, we intend to benchmark it against concurrent bugs found in real-world applications, representative of their possible flavours.

Regarding applications, we intend to use both I/O- and CPU-bounded ones, in which concurrency handles different kinds of events and is handled by different kinds of mechanisms. We start with I/O-bounded applications, which will include: (a) the web server *Tomcat*, stressed by the *SPECweb2009* web benchmark; (b) the database server *Derby*, stressed by the *PolePosition* database benchmark; and (c) the IDE *Eclipse* during normal development work. The first two are server applications, which use concurrency to handle client requests, potentially spawning hundreds of simultaneous threads, while the third is a desktop application, which uses concurrency mainly to deal with GUI events.

For CPU-bounded workloads we turn to selected multi-threaded applications from the *DaCapo* [3] and *Java Grande* [38] benchmark suites. These should employ different kinds of synchronization between threads, from barrier-based, to no synchronization apart from creation and termination of workers.

As for bugs, the evaluation should focus on all of the most prevalent kinds of concurrent bugs [21]: (a) deadlocks, which occur when two or more operations circularly wait for each other to release some resource (e.g. lock); (b) atomicity violations, which occur when there is no enforcement of the atomicity of a code region that is intended to be atomic; and (c) order violations, which occur when the order between two groups of memory accesses is flipped in relation to what is intended. For the real-world applications, the bugs should be real and we intend to find them by searching the applications' bug databases. For the benchmarks, however, they might have to be manually inserted into the programs.

The quantitative assessment during this evaluation phase should take the following metrics into account:

- Production-run recording time overhead relative to bare execution of the program, measured as the percentage increment in execution time and average CPU load increase;
- Production-run recording space overhead, both in memory and disk, measured in number of recording points and trace file size increase per time unit;
- Percentage of bugs successfully reproduced within a certain amount of replay attempts;
- Average number of replay attempts and time taken to reproduce each successfully replayed bug;

– Time overhead imposed during the 100% successful replays, measured as the percentage increment in execution time relative to native execution.

These measures should be taken in perspective of the following dimensions: number of threads; number of shared memory accesses and synchronization operations; bug type/description; and synchronization mechanism. The number of processors should also vary, so that the scalability of the approach can be assessed.

6 Conclusion

In this document we have mainly surveyed previous work on deterministic replay systems. To better understand the field, we proposed a new taxonomy that attempts to identify the vectors through which these systems can move. We take a few key conclusions from the review of previous work in deterministic replay: (a) support for input or memory non-determinism is the most distinguishing criterion, because the systems on both sides use very distinct techniques; (b) input non-determinism can be replayed efficiently with software-only solutions; (c) full handling of memory non-determinism has been proven to be efficient with hardware support; (d) software-based approaches struggle with recording data races and many avoid them altogether, recording solely synchronization races and, thus, being unable to simultaneously support multiprocessor executions and imperfectly synchronized programs; (e) recent probabilistic software-only approaches have shown potential for enabling efficient recording while supporting multiprocessors and imperfectly synchronized programs, at the cost of higher replay overhead.

Taking these lessons into account, we set off to create a deterministic record and replay system for the Java execution environment by specifying our objectives, proposing an architecture and describing an evaluation methodology for the developed system. Our main objectives are efficient production-run recording, support for multiprocessors, for imperfectly synchronized programs and ease of deployment. Since probabilistic approaches are the state of the art to achieve these objectives with a software-based solution, we decided to employ similar techniques. Synchronization races are to be recorded efficiently during production runs, while an offline state-exploration stage derives the outcomes of non-recorded events and enables deterministic replay thereafter.

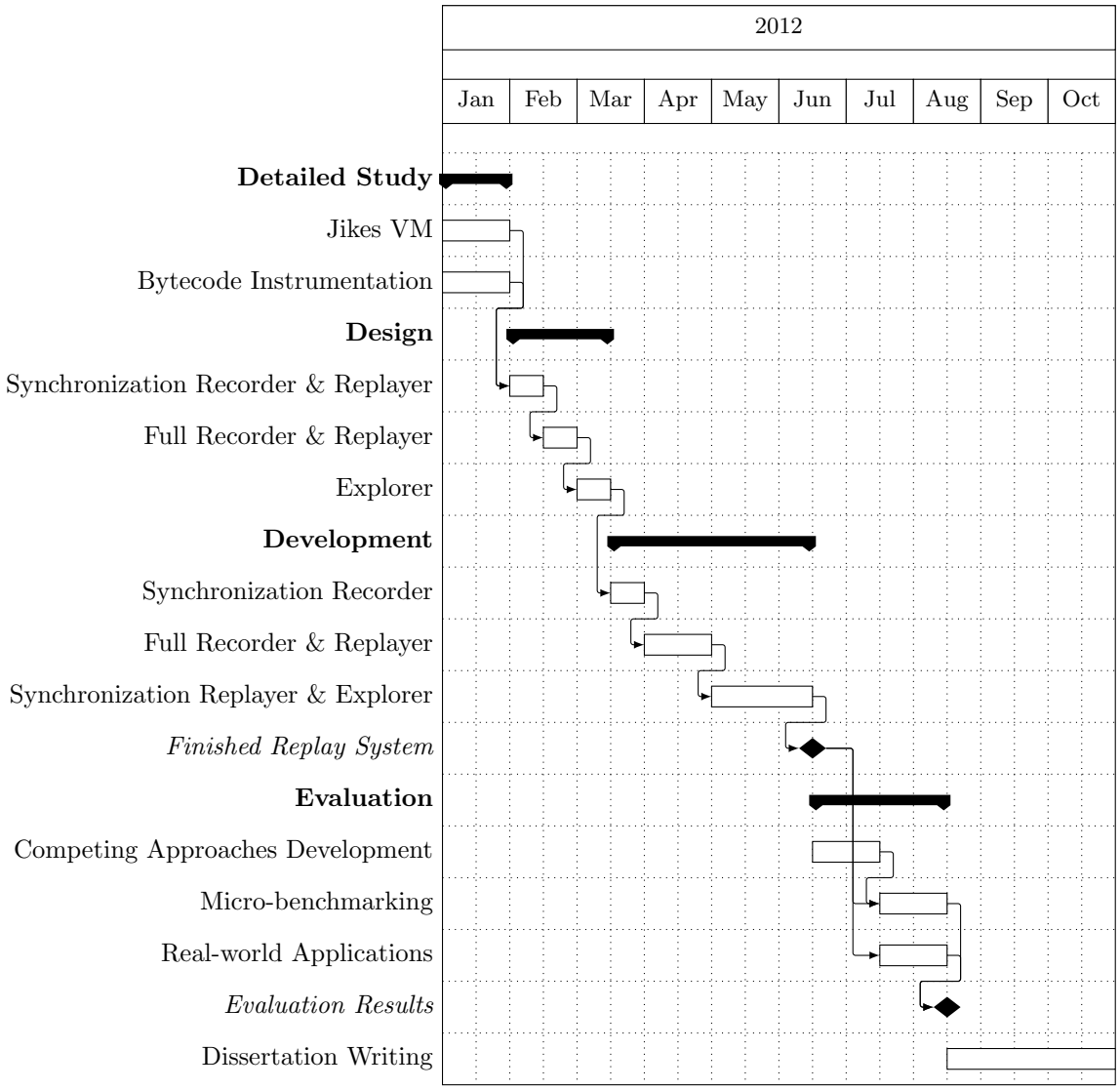
We hope our overview of previous work and proposed taxonomy enables a better understanding of the research field of deterministic replay and aspire to create a state of the art record and replay system that achieves all our objectives.

References

1. Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 193–206. SOSP '09, ACM (2009)
2. Bacon, D.F., Goldstein, S.C.: Hardware-assisted replay of multiprocessor programs. In: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging. pp. 194–206. PADD '91, ACM (1991)
3. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. SIGPLAN Not. 41, 169–190 (October 2006)
4. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. ACM Trans. Comput. Syst. 14, 80–107 (February 1996)
5. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools. pp. 48–59. SPDT '98, ACM (1998)
6. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., Bosschere, K.D.: A taxonomy of execution replay systems. In: In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet (2003)
7. Dionne, C., Feeley, M., Desbiens, J.: A taxonomy of distributed debuggers based on execution replay. In: In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (1996)
8. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. 36, 211–224 (December 2002)
9. Dunlap, G.W., Lucchetti, D.G., Fetterman, M.A., Chen, P.M.: Execution replay of multiprocessor virtual machines. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 121–130. VEE '08, ACM (2008)
10. Geels, D., Altekar, G., Shenker, S., Stoica, I.: Replay debugging for distributed applications. In: Proceedings of the annual conference on USENIX '06 Annual Technical Conference. pp. 27–27. USENIX Association (2006)
11. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: Jarec: a portable record/replay environment for multi-threaded java applications. Softw. Pract. Exper. 34, 523–547 (May 2004)
12. Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M.F., Zhang, Z.: R2: an application-level kernel for record and replay. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. pp. 193–208. OSDI'08, USENIX Association (2008)
13. Hower, D.R., Hill, M.D.: Rerun: Exploiting episodes for lightweight memory race recording. In: Proceedings of the 35th Annual International Symposium on Computer Architecture. pp. 265–276. ISCA '08, IEEE Computer Society (2008)
14. Huang, J., Liu, P., Zhang, C.: Leap: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 207–216. FSE '10, ACM (2010)

15. Huselius, J.: Debugging parallel systems: A state of the art report. Tech. rep. (2002)
16. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: Proceedings of the annual conference on USENIX Annual Technical Conference. pp. 1–1. ATEC '05, USENIX Association (2005)
17. Konuru, R.: Deterministic replay of distributed java applications. In: In Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium. pp. 219–228 (2000)
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (July 1978)
19. Landes, T.: Dynamic vector clocks for consistent ordering of events in dynamic distributed applications. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. pp. 31–37. CSREA Press (2006)
20. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Comput. 36, 471–482 (April 1987)
21. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGOPS Oper. Syst. Rev. 42, 329–339 (March 2008)
22. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the International Workshop on Parallel and Distributed Algorithms. pp. 215–226. Elsevier B.V. (1989)
23. McDowell, C.E., Helmbold, D.P.: Debugging concurrent programs. ACM Comput. Surv. 21, 593–622 (December 1989)
24. Montesinos, P., Ceze, L., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In: Proceedings of the 35th Annual International Symposium on Computer Architecture. pp. 289–300. ISCA '08, IEEE Computer Society (2008)
25. Montesinos, P., Hicks, M., King, S.T., Torrellas, J.: Capo: a software-hardware interface for practical deterministic multiprocessor replay. In: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems. pp. 73–84. ASPLOS '09, ACM (2009)
26. Narayanasamy, S., Pereira, C., Calder, B.: Recording shared memory dependencies using strata. In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. pp. 229–240. ASPLOS-XII, ACM (2006)
27. Narayanasamy, S., Pokam, G., Calder, B.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: Proceedings of the 32nd annual international symposium on Computer Architecture. pp. 284–295. ISCA '05, IEEE Computer Society (2005)
28. Netzer, R.H.B.: Optimal tracing and replay for debugging shared-memory parallel programs. In: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging. pp. 1–11. PADD '93, ACM (1993)
29. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 177–192. SOSP '09, ACM (2009)
30. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: transparent checkpointing under unix. In: Proceedings of the USENIX 1995 Technical Conference Proceedings. pp. 18–18. TCON'95, USENIX Association (1995)
31. Pokam, G., Pereira, C., Danne, K., Yang, L., King, S., Torrellas, J.: Hardware and software approaches for deterministic multi-processor replay of concurrent programs. Intel®Technology Journal 13 (2009)
32. Ronsse, M., De Bosschere, K., Chassin de Kergommeaux, J.: Execution replay and debugging. eprint arXiv:cs/0011006 (Nov 2000)
33. Ronsse, M., De Bosschere, K.: Recplay: a fully integrated practical record/replay system. ACM Trans. Comput. Syst. 17, 133–152 (May 1999)
34. Russinovich, M., Cogswell, B.: Replay for concurrent non-deterministic shared-memory applications. In: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation. pp. 258–266. PLDI '96, ACM (1996)
35. Saito, Y.: Jockey: a user-space library for record-replay debugging. In: Proceedings of the sixth international symposium on Automated analysis-driven debugging. pp. 69–76. AADEBUG'05, ACM (2005)
36. Simo, J., Garrochinho, T., Veiga, L.: A checkpointing-enabled and resource-aware java vm for efficient and robust e-science applications in grid environments. Concurrency and Computation: Practice and Experience preview (2012)
37. Slye, J.H., Elnozahy, E.N.: Supporting nondeterministic execution in fault-tolerant systems. In: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96). pp. 250–. FTCS '96, IEEE Computer Society (1996)
38. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM). pp. 8–8. Supercomputing '01, ACM (2001)
39. Sorin, D.J., Martin, M.M.K., Hill, M.D., Wood, D.A.: Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. SIGARCH Comput. Archit. News 30, 123–134 (May 2002)
40. Srinivasan, S.M., Kandula, S., Andrews, C.R., Zhou, Y.: Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In: Proceedings of the annual conference on USENIX Annual Technical Conference. pp. 3–3. ATEC '04, USENIX Association (2004)
41. Steven, J., Chandra, P., Fleck, B., Podgurski, A.: jrapture: A capture/replay tool for observation-based testing. In: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis. pp. 158–167. ISSTA '00, ACM (2000)
42. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In: Proceedings of the 30th annual international symposium on Computer architecture. pp. 122–135. ISCA '03, ACM (2003)
43. Xu, M., Hill, M.D., Bodik, R.: A regulated transitive reduction (rtr) for longer memory race recording. In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. pp. 49–60. ASPLOS-XII, ACM (2006)
44. Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B., Inc, V.: Retrace: Collecting execution trace with virtual machine deterministic replay. In: In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS (2007)

A Planning


2012
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Detailed Study
Jikes VM
Bytecode Instrumentation
Design
Synchronization Recorder & Replayer
Full Recorder & Replayer
Explorer
Development
Synchronization Recorder
Full Recorder & Replayer
Synchronization Replayer & Explorer
Finished Replay System
Evaluation
Competing Approaches Development
Micro-benchmarking
Real-world Applications
Evaluation Results
Dissertation Writing