A Taxonomy of Execution Replay Systems

Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, Koen De Bosschere

Department of Electronics and Information Systems
Ghent University
kdb@elis.ugent.be

***Abstract*— Debugging a faulty program can be very hard and time-consuming. The programmer usually reexecutes his program, while zooming in on the root cause of the bug. However, sometimes bugs seem to appear only intermittently, making it even harder for the programmer to solve them. The main reason for this is that when executing a program, there are numerous non-deterministic events taking place within the computer, which can wreck even a very carefully crafted debugging session. To deal with these ghostly bugs, one needs to remove the non-determinism from the execution and its environment. Therefore various so-called execution replay systems have been devised, each with their merits and limitations. We give an overview of the terminology used when discussing execution replay, of the causes of non-determinism within a computer, and of the current state of the art in execution replay systems.**

I. Introduction

If a cat tries to catch a mouse she'll first immobilise it with her paw effectively rendering it helpless and only then go for the kill. The reason is simple: the mouse is too fast to kill when running around freely.

The same two-phased approach is often needed during the debugging of software. The first thing that needs to be done when a bug is detected is to make sure that it can be reproduced at will. Only then can we start looking for the root cause of the bug.

The fundamental reason why this approach is necessary is that we often use "cyclic debugging" to locate the cause of bugs. When using cyclic debugging we first coarsely analyse a run of a buggy application and based on that we make a few assumptions about the cause of a bug. Then we rerun the application, maybe adding a few diagnostic statements, and observe more closely specific areas of its execution. This allows us to refine our assumptions and run the application over and over until we finally locate the root cause of the bug.

This approach can be seriously jeopardised if we are dealing with a "non-deterministic" bug, i.e. a bug that from one run to another does not always manifest itself. This will usually cause our debugging efforts to be totally thrown off track.

The way to tackle the problem of non-determinism is by using a technique called "execution replay". Execution replay consists of two phases. During the first phase, the "record phase", information about all non-deterministic events are recorded into a trace file. Then during the second phase, the "replay phase", we consult the trace file each time we encounter a non-deterministic event in order to make sure it exhibits the same effects as were seen during the record phase.

In order for an execution replay system to be applicable in practice, it needs to satisfy a number of properties:

Accuracy It must assure a replay that resembles the original execution as closely as possible.

Non-intrusiveness The intrusion of the record phase must be as small as possible to avoid so-called "Heisenbugs", which may disappear due to the intrusion we provoke. We must make certain that we record a "typical" execution and not one that has been perturbed substantially due to the record/replay mechanism.

Space efficient The space overhead of the trace file must be minimised. Again this is important from an intrusion standpoint but also from a practical standpoint: it is all too easy to generate trace files of hundreds of megabytes.

Time efficient It's important that the replay phase can be executed at a speed comparable to the original execution.

If these goals are met then we have an excellent debugging tool at our disposal. Indeed, during the record phase we can observe the behaviour of the application with little or no intrusion while during the replay phase we can use even the most intrusive debugging techniques (barring changing the state or the code of the application) while still having the guarantee that we can see the bugs as they actually happened during the original execution.

During previous years many execution replay techniques have been proposed, all having their own limitations. In particular, most execution replay approaches focus on a specific form of non-determinism but cannot handle others. It is the goal of this article to give an overview of the existing execution replay techniques and terminology and show their strengths and weaknesses. This overview will show a number of areas where further research is necessary in order to produce a complete solution for the execution replay problem.

II. Ordering and content-based execution replay

Ideally, we would like to go back in time and observe the execution of our application in its original setting. That way, all details of the execution would be exactly reproduced. Time travel however is not yet an option so we must be content with an approximation of the original execution. This means that we need to consider carefully which aspects of the execution

we consider important and wish to see replayed and which aspects can safely be ignored. Trying to find a right balance between the accuracy of the execution replay system and its intrusion is therefore a question of finding the right *level of abstraction* at which to operate.

In this article we will describe systems that reproduce the execution of an application at a fairly high level. When debugging an application we are primarily trying to reproduce the instruction stream of that application and the resulting modifications to the application's state. Sometimes, we wish to go further and reproduce the behaviour of the whole (software) system including the operating system. In this article we will assume that other (mainly hardware) aspects such as the behaviour of the bus, the cache, the movement of the head of the disks, ... do not concern us, so they do not need to be replayed exactly.

How do we go about reproducing this instruction stream? A first approach is called "content-based" replay. The idea is simple: during the record phase store all the data that is read by the instructions (from the register file and the main memory) into a trace file. During replay use this trace file to provide every instruction with the necessary input. This approach is mainly of theoretical importance since in practice it will result in enormous trace files.

Notice that the "time travel" approach mentioned earlier requires no trace file. We just need to know exactly to which point in time we have to return and what the initial state of our application was. The reason for this is that with the time travel approach we are able to let the environment behave identically to the first execution. We achieved this (in theory) by forcing the environment back into the original state and as a consequence force the ordering of the interactions between the environment and the application to happen identically. This approach is called "ordering-based" replay.

In practice we will not "time travel" but we will set up the program's runtime environment in an initial state which is almost equivalent to the original state. Then, while the program is reexecuting, we periodically nudge it in the right direction each time there is a risk that its execution would diverge from the original one.

The main advantage of this approach is that it is the application *itself* that reproduces most of the data that is read by its instruction stream. This presents us with an interesting trade-off. If we can start execution from a state that is almost identical to the original state then little additional trace data needs to be recorded. This of course requires a large amount of data to store the initial state. An inaccurate initial state on the other hand will require more trace data to keep the execution on track but less data to store the initial state.

Ordering-based replay has a large disadvantage compared to content-based replay: we are no longer able to execute instructions in isolation. Ordering-based replay requires that all the interactions with the environment actually take place so that the internal state of the application and the state of the environment are updated correctly. If this does not happen, sooner or later the environment or the application will start to diverge from their original behaviour. Content-based execution replay does not have this shortcoming. Since all the data to perform any instruction is always available, we can accurately reexecute any set of instructions in any order we desire.

Both the pure content-based and pure ordering-based approaches are not feasible in practice because they operate on a level of abstraction that is too low and therefore requires too much trace data. However, if we raise the level of abstraction, hybrid forms combining ordering-based and content-based execution replay can be very successful. The idea is to try to force some parts of the environment to reproduce the data needed for executing the instruction stream (ordering-based) and providing the rest of the data from the trace file (content-based).

III. Exploiting sequential execution

As mentioned before, a pure content-based replay scheme is not feasible, given the very large trace bandwidth and storage required to allow a replay of the execution.[1] The following observation can lead us a first step away from a pure content-based execution replay approach and move us toward an ordering-based scheme.

Processors execute instructions in a sequential manner, so most of the data consumed by an instruction has been produced by a previous instruction executed on the same processor. The result of these instructions is usually stored either in one or more registers, or in memory. This means that, if we succeed in forcing instructions to be reexecuted from a well-known state of both processor and memory, subsequent instructions will likely read the same values from the registers or from memory.

Thus, instead of restoring the local environment that is used by every instruction, as it is done in a pure content-based approach, we can exploit the fact that memory serves as a container of state information. Clearly, this saves us from recording the input data of every single instruction.

By doing this, we move the boundary at which we trace the input away from the instructions executed on the processor toward the memory system. As such, we no longer trace all the input that is consumed by individual instructions, but instead we record the content of the external influences on the memory system.

This is a general trend in execution replay: one tries to move the input boundary so that a minimum of input needs to be recorded. The price is that one needs to replay more of the environment and store more ordering information about the way in which the events within the replayed system interact with each other.

[1] Some work has been done regarding lossless compaction of execution traces [Sam89], [JH94], [JHBZ01], e.g. as required in trace driven simulations.

IV. Non-determinism in the memory subsystem

Simply reexecuting the instructions of an application from its start will usually not provide us with an accurate replay of an execution of this application. While it is true that an application will often read values from memory that it itself stored there in the first place, there are still "external interferences" that can modify such values. When such a modified value is read without taking any precautions, the reexecution will diverge and will no longer be accurate.

The changes of the memory system, the caches and the register file caused by external influences still need to be recorded. The following external influences can be identified in modern computer systems:

Dedicated input instructions Many processors have specific machine instructions for acquiring input. These instructions communicate with peripherals, e.g. the parallel ports, mice, keyboards. They typically alter the content of one or more registers. Input instructions happen synchronously, i.e. the alteration of the memory state happens at a known time and is explicitly requested by the application.

Interrupts A processor can be made aware of external events through the occurrence of interrupts. An interrupt typically alters the content of the program counter register (PC), in order to transfer control to an interrupt handler. Furthermore, it can also modify the status register and some other registers. Interrupts (except software interrupts) happen asynchronously, i.e. the time at which they alter the memory state is usually not determined by the application.

Interaction with other processors Usually there are multiple processors modifying the content of the main memory. They range from simple direct memory access (DMA) processors to memory mapped I/O, such as video memory. Apart from such systems, there exists also "true" multi-processor systems, with multiple main processors interacting for example via main memory or even directly by means of cache-snooping protocols. Here too, the actions of other processors often happen asynchronously.

If we can exploit sequential execution, we still have to deal with these three categories of interference. In general, the approach taken will be one of the following. First, one can record the contents of the input as well as the exact time at which the input becomes available in the case of asynchronous input. Second, one can extend the amount of the environment that will be reexecuted in order to obtain the correct input and therefore increase the usage of ordering-based execution replay. In the following sections we look at existing approaches for dealing with the aforementioned problems.

A. Dealing with dedicated input instructions

When dealing with input, the level of abstraction considered is important. We distinguish two levels on which input consumed by a program can be traced: (i) the level of the individual instructions, and (ii) the level of the executive or operating system, when such a system is present.

1) Defining dedicated input instructions: Here we position ourselves on the level of the individual machine instructions. Dedicated input instructions can be distinguished from regular instructions by the fact that dedicated input instructions deliver non-deterministic results, whereas regular instructions — when given the program's internal state — always deliver the same deterministic result. Despite the fact that both kinds of instructions are executed in a synchronous manner, from a program's point of view, dedicated input instructions use additional input besides the internal state of the program. We will refer to this type of dedicated input instructions as *hardware input instructions*.

Two examples are (i) the i386 instruction `IN AL,DX` for reading an input byte from I/O address in `DX` into `AL`, and (ii) the `RDTSC` instruction on a Pentium processor, which reads the timestamp counter.

On a higher level of abstraction, we find instructions that are commonly called *system calls*. These are a form of dedicated input instructions as they use information other than the program's internal state. This is the case when the internal state of the program does not include the entire content of the machine's main memory, i.e. in the presence of an executive or operating system. An example of such an instruction is the `int 0x80` instruction that can be found in binaries compiled for Linux on the IA-32 platform or the `int 0x2e` instruction in binaries for the Win32 IA-32 platform. System call instructions are of extreme importance to a program because they often are a major source of synchronous input. Moreover, because modern processors provide protection against the usage of dedicated input instructions by offering different run levels, most operating systems do not allow direct hardware I/O (as discussed in the previous paragraphs), but offer an ABI by which programs can communicate with the outside world. If a program tries to perform direct hardware I/O while in the user-mode run level, the processor raises an exception.

Finally, some of the dedicated input instructions are a combination of the above two categories, i.e. some system calls retrieve their inputs from both the OS kernel memory and the outside world through hardware input instructions.

2) Replaying dedicated input instructions: Both the content-based and the ordering-based approaches could be used to achieve the goal of replaying dedicated input instructions.

a) Replaying hardware input instructions: This is only feasible using a content-based approach. Ordering-based replaying the input caused by external hardware implies reconstructing the electrical signals that these components generate,

which obviously has nothing to do with software execution replay. There are two possible ways to reproduce the result of hardware input instructions: (i) the execution replay system instruments the instructions found in the binary, either statically or dynamically [RDB01], [MRDB02], or (ii) the execution replay system relies on processor (or executive) exceptions to deal with them. In the latter case, the execution replay system needs to provide an adequate handler to save the input values during the record phase and to restore them during the replay phase.

b) Replaying system calls: Analogous comments can be made for execution replay of system call instructions. When we try a pure ordering-based approach, this would require an execution replay of the entire system (including the executive or operating system). As a consequence, the execution replay system no longer makes a distinction between the user program and the operating system. This annihilates obviously the characteristics of system calls, since the term can only make sense when there is a boundary between the executive or operating system and the user program.

Clearly, execution replay of system calls basically requires a content-based approach so all changes made by a system call to a program's memory can be identified and traced.

Although this seems fairly straightforward at first sight, there are some system call peculiarities that must be dealt with:

Detecting the memory regions changed by system calls Although most system calls change well-known memory regions, some system calls write to memory regions that are difficult to discover from the execution replay tool's point of view. An example of such a system call is `ioctl` which offers an extensible interface where the specific implementation is usually provided by add-on device drivers. Keeping up with all device drivers in order to chart every memory region that can be changed by system calls like `ioctl` is virtually impossible. Thus we need other means to detect the memory regions changed by such system calls.

Replaying the in-kernel side effects of system calls It is not possible to replay system calls using just a pure content-based approach. Some system calls must actually reoccur during the replay phase, e.g. the system call handling `printf`, to avoid so called black-hole effects where all the observable actions of a program are lost. Other system calls need to be reexecuted because they have side effects within the OS kernel. For example, within the Linux kernel, the system call `mmap` is used for user memory allocations. Obviously, if memory is not allocated during the replay phase, the kernel will most likely throw an exception when the application tries to access the memory region it has supposedly allocated. Thus, to maintain overall consistency, some system calls simply have to be reexecuted.

3) Existing execution replay systems for dealing with program input: As mentioned before, on the lowest level of the machine instructions, only a content-based approach can be used. To our knowledge, no tool exists that deals with this. There exist however systems that deal with input on a higher level of abstraction.

a) TORNADO: The TORNADO tool [CRDB03] enables one to replay the input produced by system calls, while leaving other sources of non-determinism untreated. It requires no instrumentation, recompilation or relinking of the binary. TORNADO runs on the Linux IA-32 platform. It is capable of replaying applications with an overhead of less than a factor 2 (during both record and replay phases).

TORNADO uses the following technique to allow an accurate replay of a program execution. During the record phase it intercepts all the system calls in order to trace changes made to userland memory. During replay, it reapplies all the recorded changes to the userland memory pages. Hence, TORNADO primarily uses a content-based approach to reach its goals. However, TORNADO is also capable of dealing with the problems identified in section IV-A.2.

To (non-naively) handle the detection of memory regions changed by system calls TORNADO uses an altered Linux kernel. This kernel has instrumented to-userland write primitives which allows one to trace every write operation performed by system calls to userland memory. When a system call is executed, an in-kernel linked list is constructed containing the information about where and how much data the system call writes to the userland memory. The execution replay tool can consult this list afterwards to collect the needed information to allow a deterministic replay. Managing the list containing the write information is done by extending the `ptrace` system call.

Another interesting technique to trace memory write operations (performed by system calls), namely *Compressed Differences* [PXN95], is using the virtual memory access protection facilities the OS offers. With this technique the tool provides a memory page fault handler, allowing it to compare original memory pages with copy-on-write pages that have been changed. Still, after detecting a changed page, this technique requires scanning the memory pages to find changes because storing the entire page in the trace is unfeasible.

b) jRapture: jRapture [SCFP00] is a tool designed to capture interactions between a Java application and the underlying system, i.e. interactions with e.g. the user interface, files, keyboard, etc. During the replay phase, it presents the executing threads with the same input sequence as they received during the record phase. As such, jRapture clearly places itself between the JVM and the application the former is executing. In [SCFP00], three ways are distinguished in which the state of a process can be modified by a method call: (i) by returning a value, (ii) by changing the values of its parameters, and (iii) by changing the fields accessible to the method. Steven et al. modified the Java API classes that

interact with the JVM and the underlying system through JNI calls. This clearly deals with (i) and (ii) stated above, but leaves point (iii) untreated. From speaking with the authors, we gathered that the latter point was planned for the future.

Even for (i) and (ii), some difficulties need to be tackled. First of all, so called primitive values (e.g. int, float, ...) can be written to a trace file immediately. However, objects are an entirely other matter. Their state can only be written to a trace file if the classes from which they are instantiated implement the *Serializable* interface. Clearly, for any given application, this is not always the case.

Another problem arising when dealing with user interface interactions that need to be recorded, is the event model the Java AWT uses. Clearly, the source generating an event must be recorded. Moreover, when replaying, the recorded source must be mapped onto the object present in the replayed execution. To deal with this, jRapture identifies objects using the id of the thread that created them and a sequence number. Threads are identified by their parent thread id and a sequence number.

We have found no real indication of the slowdown an application incurs when running on a jRapture platform. The authors claim that for small application, the slowdown was limited to 0.2% in time, while for replaying a small user interface the slowdown was of the order of 1% during record and 0.04% during replay. No usable figures on trace file size are given.

B. Dealing with interrupts

The problem that one needs to tackle when dealing with replaying interrupts is to determine the exact timing at which the (asynchronous) interrupts take place. Tracing the program counter and the wall clock simply is not sufficient; in a loop (or with multiple calls to a function) multiple instances of the same instruction get executed. This makes tracing only the program counter useless; we need to trace the timing as well. Since the execution speed can vary between different program instances (e.g. because of OS activity), the wall clock cannot be used to represent the execution speed/duration of instructions. Thus one needs a *logical clock*. Such a clock can be constructed by using a software instruction counter [MCL89], or by using a hardware instruction counter [CL87], or by using an interpreter [FB88].

As it is the case for dedicated input instructions, one needs to keep the level of abstraction in consideration when dealing with the replay of interrupts. Dealing with IRQ's can be regarded as low level interrupt replay. Analogous, one can label scheduling replay as interrupt replay at a higher level of abstraction.

In the next paragraphs we will discuss some systems that are capable of replaying interrupts.

1) Interrupt Replay: In [AL95] Audenaert and Levrouw describe a method for replaying parallel programs on bus-based shared-memory multiprocessor systems that use interrupts, as an extension to Instant Replay [LMC87] (see Section IV-C.2.e). The system deals with both the user program and the OS. The relative positions of the interrupts with respect to the interrupt window[2] are traced and reenforced during replay. The tool requires that the following assumptions are true: (i) hardware interrupts can be disabled separately, either by the PIC[3], or by a simulation in software, (ii) the environment can be simulated (by an external process), and (iii) there are no interrupt races, i.e. a data race between the interrupt service routine and the foreground routine.

The goals of Interrupt Replay are twofold. First of all, the interrupts generated by the program, such as 'traps' and 'int' instructions are replayed by calling the interrupt service routine. This includes hardware generated interrupts that can be simulated in software. Second, those interrupts that cannot be simulated, e.g. because the processor is interrupted by another processor, are effectively regenerated during replay.

To determine the timing of the interrupts, (approximate) software instruction counters [MCL89] are used. Only the relevant instructions, i.e. the explicit and implicit interrupt enable/disable instructions of both the CPU and the PIC, the synchronisation operations, and the instructions generating interrupts are counted.

By using a clever encoding, the space overhead is about the same as that of Instant Replay, except for the additional log entries for the interrupts. The time overhead is somewhat, but not significantly, larger than for Instant Replay, due to software instruction counter management.

Interrupt Replay has been implemented on a Modula-2 machine.

2) Repeatable scheduling: The Repeatable Scheduling Algorithm (RSA) [RC96] focuses on the replay of application scheduling on uniprocessor systems. The technique tackles the problems regarding non-determinism introduced by shared memory very efficiently by exploiting the restriction to uniprocessor systems. During the record phase preemption information about the threads of the traced application is saved. During the replay phase the scheduler is forced to make the same scheduling decisions as during the original run. By reapplying the same scheduling to an application, RSA assures that all accesses to shared memory (within the same application) will reoccur as during the original run. This requires RSA to have a means to count the number of executed instructions on a per-thread basis.

The technique is language independent, because it does not need to know which instructions access shared memory and which are limited to private data access. On the other hand, the implementation on the Mach 3.0/UX OS requires compile-time instrumentation of the original program and system libraries augmenting them with a (user space) software instruction

[2]This is the timespan in which a processor has not disabled interrupts.
[3]Processor Interrupt Controller.

counter [MCL89] using a dedicated (Pentium) processor register. Furthermore the thread scheduler must be modified to inform the recording system about upcoming thread switches and to be able to read thread switch information out of the trace file.

The instruction counter is only incremented each time the application makes a backward control transfer or a function call. Thus, to identify each instruction in the execution sequence of a thread uniquely, the $\langle \text{counter value}, \text{instruction pointer} \rangle$ pair is sufficient.

Such pairs are recorded in the record phase. During a replayed execution the counter and the placement of breakpoints are used to reapply the original scheduling. [RC96] has implemented this using a user-space scheduler, although OS kernel support is also possible and should result in less performance overhead [RC95].

The approach is effective and yields a trace bandwidth in the order of 1-2 KiB/s, depending on the scheduling activities of the program. The typical performance overhead is 10-15%, this is mainly caused by the instructions added to maintain the software instruction counter.

3) DejaVu: DejaVu [CS98] is an execution replay infrastructure designed at IBM for dealing with non-determinism in Java applications caused by threads and other related concurrent constructs. Other sources of non-determinism, such as file I/O, are ignored. Later versions of the tool are also capable of handling some windowing events [CS98] and network events [KSC00].

To stay independent of the underlying thread scheduler, the notion of a logical thread schedule is introduced. Simply put, multiple physical thread schedules can map onto a single logical thread schedule. However, enough information is retained in the logical schedule to reproduce the execution behaviour during replay.

DejaVu distinguishes two event categories: (i) synchronisation operations and accesses to shared variables, called critical events, and (ii) noncritical events, that can only influence the thread that executes them. Therefore, the scheduling of noncritical events is unimportant for regenerating the recorded execution behaviour. DejaVu attaches a global scalar timestamp to each critical event, thus imposing a total ordering. To limit the size of the trace files, only the timestamp information of the first and the last event of each thread schedule interval is stored. However, the technique used is only viable on a uniprocessor system, because each critical event must be synchronised on the global timestamp. Thus, only non-critical events may actually run concurrently, leading to short thread intervals and huge trace files. On a uniprocessor, overheads range from 0 to 88% during the record phase, and less than 65% during replay. Trace files are very small, less than 1 KiB/s.

For uniprocessor systems, DejaVu has been further extended [CANS01], allowing it to replay the entire JVM, which includes the thread scheduler.

As it is the case for jRapture, DejaVu is not capable of deterministically replaying native methods (using JNI) with non-deterministic side effects, unless they are also compiled with some DejaVu-aware compiler.

C. Dealing with the interaction with other processors

1) Problem description: There are a number of possibilities for other processors to interact with a running application using shared memory.

- Multi-threading: all processors execute code that is part of the same application. In order to tackle the non-determinism introduced by unordered memory operations, one can either use content based or ordering-based replay. Content-based replay will trace the memory values read by all load operations (of course, this will introduce an intolerable overhead; [PL88] already measured up to 10MiB/s bandwidth back in 1988 and this method is therefore not feasible). Ordering-based replay will simply trace the order in which the memory operations took place. During replay, each load operation should be stalled until the store operation that writes the value we will read has been executed [LMC87].
- Multi-processing: the other processor(s) executes code from another application. If the other application has mapped some of our memory in its address space[4], the application can change these memory locations. Of course, this is only possible if the application allows other applications to map some of its pages. Using the proper memory protection for the shared pages, detecting such a load operation is not that difficult and content-based replay can therefore be used to replay such an application.
- The other processor can be a co-processor that is executing some code on our behalf, e.g. a math co-processor, a DMA-controller or a graphics card with a 3-D processor. The memory (or register) locations that will be changed by the co-processor are known, but the exact moment of this operation is not always known. Dealing with this type of non-determinism requires content-based replay, unless it is possible to let the co-processor reexecute the same code and use the same state as in the original execution. This is the case for a math or a 3-D co-processor, but not for DMA.

Until now, most research has been focusing on replaying multi-threaded programs. As stated above, this either requires content-based or ordering-based replay of all (shared) memory operations. The content-based approach requires the largest trace bandwidth. In order to reduce this overhead, techniques have been proposed to introduce ordering-based replay for a subset of all memory operations.

The problem that these techniques deal with, is the replay of shared memory accesses in multi-threaded programs; the other

[4]Called UNIX shared memory on UNIX systems.

memory accesses can be left aside by the execution replay system. The disadvantage of using an ordering-based approach lies within the detection of shared access events. One of three things are used to allow for detection of shared memory access events:

- all accesses to shared memory are well defined at either the language level or the OS level
- augmentation of the trace process to detect shared memory accesses dynamically
- all memory accesses are considered to be shared accesses

One technique is called synchronisation replay: only the synchronisation operations are replayed in the correct order. As this will only produce an equivalent reexecution in the absence of data races, the program (execution) should be checked for data races as well.

2) Existing systems:

a) RecPlay: RecPlay [RDB99] is a replay tool that uses a number of phases. During the first phase (record) the order of the synchronisation operations, as executed by an application, is traced. This phase uses the ROLT [LAV94] method for attaching timestamps from a scalar Lamport clock (a logical clock [Lam78]) to each synchronisation operation. A correct replay *in the absence of data races* is possible by stalling each synchronisation operation until all operations with a smaller timestamp have been executed. As this only works in the absence of data races, RecPlay will check the first reexecution for data races. RecPlay checks parallel pieces of code (these are detected by checking the happened-before [Lam78] relation using vector clocks [Fid89]) for data races and will warn the user if a data race has been found. If this is the case, the user can hunt for the cause of the data race by *deterministically* replaying the application and removing the data race using cyclic debugging techniques. Note that the data race only can wreak havoc *after* its occurrence, hence the part before the data race will be replayed deterministically. If no data race is found (or if the data races have been removed), the application can be replayed without checking for data races, greatly improving the replay speed.

RecPlay has been implemented on the SPARC Solaris platform. To record the synchronisation events, a just in-time instrumentation (JiTI) technique is used. Notice that this requires no adaptation of existing binaries. Instead, using a shared library, binaries are instrumented while they are executed.

The overhead of RecPlay in the Solaris implementation is very low, on average about 2.1 percent, with 26 percent in the worst case scenario during the record phase. During the replay phase without data race detection, the overhead is 91 percent on average and 570 percent in the worst case. With data race detection, the execution is about 36 times times slower, with a peak of 236 times for the worst case.

b) JaRec: JaRec [GCRDB] is an execution replay system designed to deal with some aspects of non-determinism that can occur when executing multi-threaded Java programs. JaRec tackles one problem, namely the occurrence of synchronisation races. Similar to RecPlay, the tool is used in two distinct phases (i) a record phase, during which a trace of the synchronisation operations is recorded, and (ii) a replay phase, during which the order of the recorded operations is forced onto the execution. JaRec requires a program to be data race free in order to guarantee a correct replay. The tool can be used on multi-processor systems.

JaRec operates entirely on the Java-bytecode level. Each class that is loaded by the JVM is transferred through the JVMPI to an instrumentor, after which the instrumented class is loaded. This ensures that no static instrumentation is required. The instrumentation modifies synchronised methods and blocks, invocation sites of wait and notify as well as the starting and joining of threads. As such, JaRec can be used without any modification of the virtual machine running underneath.

The overhead of JaRec ranges from 10% to 125% on microbenchmarks, while on macrobenchmarks, the observed overhead lies around 80% during the record phase. During the replay phase the overhead varies from 40% to 300%.

c) IGOR: IGOR [FB88] is a system for replaying programs, based on checkpointing techniques. Given a checkpoint, it reconstructs the state of the program at that point and allows the program to continue its execution. IGOR does not record external I/O, so the replayed execution may differ from the original execution if the environment has changed. Also, IGOR does not handle non-determinism caused by multi-threaded programs.

The tool gathers trace information on the level of individual virtual memory pages. To do this, it employs a new `pagemod()` system call, which determines the set of pages that have been changed since the previous checkpoint. The checkpoints themselves are controlled by another system call, i.e. `ualarm()`.

During the replay phase, IGOR scans the trace file to obtain the most recent checkpoint for each virtual memory page. Then forward emulation (using an interpreter) is used to continue the execution from the last checkpoint up to an instruction specified by the user.

IGOR requires one to use (i) a modified compiler to trace data allocations, (ii) a modified library, and a modified loader to start the tracer and to allow dynamic function replacement. It runs on a DUNE system [AP88], with the additional `pagemod()` and `ualarm()` system calls. At record time, the overhead varies from 50% up to 400%, while during replay the execution is about 140 times slower.

An interesting technique to reduce the trace data for improving incremental checkpointing is *Compressed Differences* [PXN95]. This technique only traces the differences between the altered pages.

A major drawback of incremental check-points remains that they trace orders of magnitude more data than necessary as

shown later by Netzer [NW94].

d) Recap: Recap [PL88] is a tool that provides the illusion of reverse execution for parallel programs. It uses a combination of checkpointing and replay to provide the user with an illusion of reverse execution. Using a content-based approach it is capable of replaying the result of system calls and shared memory reads, as well as the times that asynchronous events (signals) occur. The checkpointing is done by forking (copy-on-write) the replaying process periodically. The shared memory reads get traced using compile-time instrumentation (for the Allegro programming environment), which is the biggest performance bottleneck of Recap. The system calls get traced using a special Recap run-time library. Recap also tries to replay signals, but fails to replay the exact timing when the signals occurred (only tracing the program counter and the wall clock is not sufficient).

The main disadvantage of this technique is the amount of trace bandwidth needed to allow for an accurate replay. Even for a slow processor (VAX-11/780) 1 MiB/s of trace data gets generated. This drawback is handled by later tools like *Instant Replay* (see Section IV-C.2.e). Under Recap, a replay execution also runs different code than the original execution.

e) Instant Replay: Instant Replay [LMC87] is a technique to replay shared memory accesses using an ordering-based approach. This technique restricts the program to access shared memory objects only through well-defined CREW (for concurrent-reader-exclusive-writer) protocol primitives, which it instruments for the execution replay. Each shared memory object is augmented with a version number. This version number is incremented after each write access (during both record and replay phase). Each thread records versioning information to its own trace file.

During the record phase a reader traces the current version number of its shared object. A writer traces the current version number of its shared object and the number of readers since the previous write access on his shared object.

During the replay phase a reader stalls until the current version number of its shared object matches the previously traced version number. A writer blocks until the version number on its shared object matches the previously traced version number and until the number of readers also matches the traced count.

Instant Replay was implemented for the Chrysalis OS on the Butterfly parallel processor by instrumenting the OS provided primitives for accessing shared data types. The technique tends to require huge amounts of trace data when the shared memory access granularity within the program is very small. On systems where every memory access can be considered as a shared memory access this technique is virtually useless.

The same technique was later implemented for a cyclic debugger for the pSather programming language [Min94].

f) Netzer's approach: An optimal trace procedure for shared-memory accesses has been proposed by Netzer [Net93]. This algorithm detects data race conditions on-the-fly between different shared-memory accesses. By only tracing these racing shared-memory accesses a minimal amount of trace bandwidth is consumed to allow for a deterministic replay. Thus this algorithm determines the minimal sequencing information and voids transitive ordered access information. The algorithm depends on vector timestamps [Fid89] for each thread and for each shared object. As this method requires the interception of all memory operations, a substantial runtime penalty can be expected. Each vector requires to have as much entries as there are threads (making it difficult to handle dynamically created threads). For each shared-memory access the vector for that thread and the vector for the shared memory object must be updated (expensive). By comparing the vector timestamps the dependence graph actually gets dynamically constructed. Whenever a new edge is detected this ordering is being traced since it denotes a data race condition.

V. Conclusions

In Table I we give an overview of all the tools previously discussed. The top section of the table contains the different types of non-determinism that record/replay tools try to eliminate. The bottom section of the table lists *positive* properties. Thus a ✔ can be thought of as a good property for the corresponding system (from the record/replay's viewpoint). The absence of certain properties for a record/replay system does not directly imply this system is less suited for replaying its addressed type(s) of non-determinism. However, some properties have a larger impact on the usability of the system than others. Especially the requirement of a *shared object access protocol* is important, since this often determines whether a system is capable of replaying data races —besides regular synchronisation races— or not.

As depicted in Table I, none of the current record/replay systems are capable of handling every form of non-determinism as would be required for perfect cyclic debugging. It is our opinion that this has nothing to do with some sort of mutual exclusion problem inherent to the different proposed solutions to handle the non-determinism. It is merely a matter of lack of solid implementation of record/replay systems. Since most of these systems emerged out of scientific research they all tend to address only one particular type of non-determinism, leaving the other types untouched.

Acknowledgements

Frank Cornelis and Andy Georges are supported by the Promotion of the Scientific-Technological Research in the Industry (IWT), on the SEESCOA research project. Michiel Ronsse, Mark Christiaens, and Tom Ghesquiere are supported by the Special Research Fund from Ghent University.

The authors would like to thank Jonas Maebe for the frequent discussions on the subject of this paper.

		DejaVu	DejaVu (Jalapeño)	IGOR	Instant Replay	Interrupt Replay	JaRec	jRapture	Recap	RecPlay	RSA	Tornado
Non-determinism	Hardware input instructions											
	System calls							✔	✔			✔
	Interrupts (scheduling)		✔			✔					✔	
	Shared-memory (content-based approach)			✔					✔			
	Shared-memory (ordering-based approach)	✔			✔		✔			✔	✔	
Properties	No compile-time instrumentation required	✔	✔		✔		✔	✔		✔		✔
	No link-time instrumentation required	✔	✔		✔	✔	✔	✔		✔	✔	✔
	No load-time instrumentation required	✔	✔		✔	✔			✔	✔	✔	✔
	No run-time instrumentation required	✔		✔	✔	✔	✔	✔	✔		✔	✔
	No OS/executive instrumentation required					✔	✔	✔	✔	✔	✔	
	Language independent				✔					✔	✔	✔
	Shared object access protocol independent		✔	✔							✔	✔
	Provides multiprocessor support	✔			✔	✔	✔		✔	✔		
	No preemptive restrictions		✔	✔	✔	✔		✔	✔	✔	✔	✔
	Hardware independent	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

TABLE I
EXECUTION REPLAY SYSTEMS OVERVIEW

REFERENCES

[AL95] K. M. R. Audenaert and L. J. Levrouw. Interrupt replay: a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10):601–612, December 1995.

[AP88] J.L. Alberi and M.F. Pucci. The dune distributed operating system. In *Proceedings of the First Using National Conference*, September 1988.

[CANS01] J.-D. Choi, B. Alpern, T. Ngo, and M. Sridharan. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[CL87] T. A. Cargill and B. N. Locanthi. Cheap hardware support for software debugging and profiling. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 22 of *SIGPLAN Notices*, pages 82–83, October 1987.

[CRDB03] F. Cornelis, M. Ronsse, and K. De Bosschere. TORNADO: A novel input replay tool. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '03)*, Las Vegas, Nevada, USA, June 23–26 2003. CSREA Press.

[CS98] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *ACM Sigmetrics Symposium on Parallel and Distributed Tools SPDT98*, pages 48–59. ACM, August 1998.

[FB88] S.I. Feldman and C.B. Brown. IGOR: A system for program debugging via reversible execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, May 1988.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*, pages 183–194, January 1989.

[GCRDB] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A portable record/replay environment for multi-threaded Java applications. *Submitted to Software: Practice and Experience*.

[JH94] E. E. Johnson and J. Ha. PDATS: Lossless address trace compression for reducing file size and access time. In *Proceedings of 1994 IEEE International Phoenix Conference on Computers and Communications*, pages 213–219, April 1994.

[JHBZ01] E. E. Johnson, J. Ha, and M. Baqar Zaidi. Lossless trace compression. *IEEE Transactions on Computers*, 50(2):158–173, February 2001.

[KSC00] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pages 219–228, May 2000.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LAV94] L. J. Levrouw, K. M. Audenaert, and J. M. Van Campenhout. A new trace and replay system for shared memory programs based on Lamport Clocks. In *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, pages 471–478. IEEE Computer Society Press, January 1994.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[MCL89] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. *ACM SIGARCH Computer Architecture News*, 17(2):78–86, April 1989.

[Min94] M. Minas. Cyclic debugging for pSather, a parallel object-oriented programming language. In *Proceedings of DAGS'94 Symposium*, July 1994.

[MRDB02] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Va, September 2002.

[Net93] R.H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, May 1993.

[NW94] R.H.B. Netzer and M.H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation (PLDI)*, pages 313–325, 1994.

[PL88] D. Pan and M. Linton. Supporting Reverse Execution of Parallel Programs. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, May 1988.

[PXN95] J. Planck, J. Xu, and R. Netzer. Compressed differences: An

algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[RC95] M. Russinovich and B. Cogswell. Operating system support for replay of concurrent non-deterministic shared memory applications. In *Bulletin of the Technical Committee on Operating Systems and Applications Environments (TCOS)*, volume 7, pages 15–19. IEEE Computer Society, Winter 1995.

[RC96] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, volume 31 of *SIGPLAN Notices*, pages 258–266, Philadelphia, Pennsylvania, 5 1996.

[RDB99] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 5 1999.

[RDB01] M. Ronsse and K. De Bosschere. JiTI: A robust just in time instrumentation technique. In *Proceedings of the Workshop on Binary Translation*, volume 29, pages 43–54. ACM Press, March 2001.

[Sam89] A. D. Samples. Mache: No-loss trace compaction. In *Proceedings of the Sigmetrics 89 Conference*, pages 89–97, 1989.

[SCFP00] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167. ACM Special Interest Group on Software Engineering, ACM Press, NY, USA, 2000.