# The Case for Execution Replay using a Virtual Machine

Nicolas Loriant, Jean-Marc Menaud

# The Case for Distributed Execution Replay using a Virtual Machine

Nicolas Loriant Jean-Marc Menaud
Obasco Group, EMN-INRIA, LINA
Nantes, France
{nloriant,jmenaud}@emn.fr

## Abstract

*Debugging grid systems is complex, mainly because of the probe effect and non reproducible execution. The probe effect arises when an attempt to monitor a system changes the behavior of that system. Moreover, two executions of a distributed system with identical inputs may behave differently due to non determinism. Execution replay is a technique developed to facilitate the debugging of distributed systems: a debugger first monitors the execution of a distributed system and then replays it identically.*

*Existing approaches to execution replay only partially address the probe effect and irreproducibility problem. In this paper, we argue for execution replay of distributed systems using a virtual machine approach. The VM approach addresses the irreproducibility problem, it does not completely avoid the probe effect. Nevertheless, we believe that the full control of the virtual hardware addresses the probe issue well enough to debug distributed system errors.*

## 1. Introduction

When debugging sequential programs, developers often repeatedly execute their programs with identical inputs in order to narrow down the search for suspected errors. This method, commonly referred to as execution replay or cyclic debugging, is not viable in the context of distributed systems such as Grid applications. Indeed, the execution of distributed systems is often affected by the ordering of some events, *e.g.*, network packets arrivals. That ordering may change for two executions with identical inputs because of the non determinism of the underlying communication layers, or of the asynchronous behavior of nodes.

Execution replay in the context of a distributed system requires some support from the debugger. The debugger must first monitor the execution to capture all sources of non determinism. If the developer suspects an error, he may attempt to locate its source by arbitrary placing a breakpoint in previously executed code, causing the debugger to execute it backward. Then, the developer may replay the execution identically. The debugger does not have to replay the entire execution, which is impracticable for most grid applications, but it must be capable of unwinding the execution far enough to surround the error.

Building such a debugger poses two issues. First, the addition of monitoring code may modify the computation time or the resource usage in a way that hides the system behavior one wants to observe. This is known as the *probe effect* [8].[1] Second, certain behavior in a non deterministic system, *i.e.*, a hardware random number generator, cannot be repeated. Thus, it may not be possible to verify that a bug has been removed. This is known as the *irreproducibility effect* [12].

In this paper, we argue for an approach based on virtual machines for execution replay of distributed grid systems. This approach will address both the probe and the irreproducibility problems. First, a virtual machine may compensate for the probe effect by mimicking the real machine behavior well enough to capture distributed errors. Second, the hardware emulation layer of a virtual machine allows to reproduce non deterministic behaviors to get ride of the irreproducibility problem.

This paper is organized as follows. Section 2 recalls the types of errors that are specific to distributed programming. Section 3 presents challenges in replaying executions of distributed programs and how these are addressed by existing execution replayers. Section 4 motivates the virtual machine approach, then Section 5 outlines a possible implementation in QEMU [2]. Section 6 concludes.

## 2. Errors in distributed programming

Classic debuggers such as GDB help track errors common in sequential programing, such as null pointer dereferencing. In this section, we consider two categories of errors that are specific to distributed systems: synchronization errors and race conditions.

[1] *a.k.a.* the Heisenberg uncertainty principle.

### 2.1. Synchronization errors

A distributed system consists of a set of tasks that progress independently. In order to complete, tasks compete for resources: CPU time, data, or hardware devices. As a task does not know what resources other tasks are manipulating, the use of potentially shared resources must be protected by synchronization primitives. Locks are the most basic synchronization primitive.

Forgetting to use such primitives may lead to the interleaved execution of two or more tasks competing for a shared resource. Such misuse of a resource often leads to errors such as data inconsistency or invalid pointer dereferences. These is known as interleaved errors.

At the opposite, incautious use of locks can result in a deadlock. Consider the following situation: a task $T_a$ requires the lock $L_x$ that is held by the task $T_b$. The task $T_a$ is interrupted and $T_b$ is elected by the scheduler, $T_b$ requires the lock $L_y$ that is held by $T_a$. Both tasks are stalled waiting indefinitely for the other to release the lock it holds.

A third possibility is livelock, in which a task is stuck in a loop that does not advance the system toward its goal. For example, when two tasks wait in a loop for a condition to become true, if each cancels the other's condition just before it comes to the test, both may be forced to loop forever.

### 2.2. Race conditions

A race condition arises when two or more tasks of a system are competing for the same resource at some point during the execution. The system may experience different behavior depending on what task wins the race. Races occur frequently on a shared memory architecture, whenever two or more tasks asynchronously access a shared resource.

A classical example of a race condition is a file-system race in `Binmail`, a *setuid root* program. When `Binmail` delivers an email, it uses the `stat` system call to check the existence of a certain temporary file. If the file exists and is not empty `Binmail` truncates it using the `open` system call. A race condition may arise if one replaces the temporary file by a symbolic link to another file before the `open` system call.

Synchronization and race conditions errors are difficult to locate and analyze using classic debuggers. Execution replay is an approach that tries to facilitate the debugging of such errors.

## 3. Execution replay of distributed systems

The probe and irreproducibility effects make it difficult to replay distributed systems. This section discusses these two problems and how they are addressed by existing approaches to replay systems.

### 3.1. The Probe effect

The probe effect also known as the Heisenberg uncertainty principle [8, 16], arises when code is added/removed to a system. The modification consumes resources (CPU time, memory...) in a way that may impact on the rest of the program. Hence, the program execution may have different temporal behavior, performance, or in turn results.

Most errors in distributed programming are due to unplanned temporal interactions between the different tasks of the system. Hence, adding monitoring code may hide an error or generate errors that would not exist in the release version of the system. This makes monitoring unusable, if the developer suspects an error from a legacy execution and corrects an error generated by the probe effect. Various monitoring approaches address the probe effect by avoiding it, minimizing it, or simply ignoring it.

*Hardware* approaches avoid the probe effect by embedding monitoring devices inside the target system [15]. One approach is to record all the traffic on the system bus. This approach, however, has two major drawbacks. First, the quantity of data that is exchanged on a system bus is large, but this problem is somewhat inherent to the monitoring problem. Second, the messages sent over a bus gives only an external view of an execution. As the trend in hardware technology is to embed more and more chipsets inside a single device, this limitation is problematic. More intrusive solutions exist to probe devices, for example development kits for real-time computing offer a inner view of a processor internals. Nevertheless, these remain costly solutions that are only appropriate for specific applications.

*Software* monitoring cannot avoid the probe effect. When the monitoring code remains inside the release version of a system, the probe effect may just be ignored [14]. While one can argue this strategy causes performance degradation, the resulting "aircraft black-box" may later be analyzed when the release of the program experience a failure. Software-level monitors mainly vary in their level of monitoring. System-level monitoring has access to the internals of the operating system. It is then possible to analyze many of the low-level aspects of system performance, such as the Translation Look-aside Buffer (TLB) or cache related events. On the other hand, task-level monitoring gathers a smaller set of information that is less intrusive and easier to analyze.

### 3.2. The irreproducibility effect

The irreproducibility problem describes the fact that a certain behavior in a non deterministic system can not be replayed on command [12]. If a non deterministic behavior is the source of an observed error, it might be impossible to reproduce that behavior, or to check that the error has been removed. For example, given a kernel level monitor, one may observe an error caused by an invalid interleaved execution of two process that both acquire a shared resource. If the replay environment does not control the scheduling, the error will not be reproducible.

Replay environments primarily vary in their degree of abstraction, *i.e.*, in what is reproducible. Russinovitch et al. proposed a repeatable scheduling algorithm for a shared memory uni-processor. Their approach forces scheduling decisions to be the same during the replay as during the reference execution [10]. Kilgore and Chase considered event based systems where the ordering of messages is the only source of non determinism [7]. Their replay environment ensures identical ordering of events and allows reordering of events according to some rules in order to provoke new failures.

Shobaki and Lindh proposed a history browser to analyze events gathered by a hardware monitor [11]. Event browsing is a higher level approach than traditional debuggers whose output are sometimes too detailed.

The probe and irreproducibility effects make it difficult to design an execution replay debugger. We believe a virtual machine approach overcomes these difficulties.

## 4. VM based approach

This section first presents virtual machines and their use in distributed programming. Next, it discusses how virtual machines can be used for execution replay.

### 4.1. Virtual machines and distributed systems

Virtual machines [1, 2, 13] are execution environments that share the same real machine, *i.e.*, the host. Each environment creates the illusion of being an entire computer, *i.e.*, a target. Targets are isolated from each other. The term *virtualization* is used when the target machine implements the same "hardware" as the host machine, whereas a SPARC target running on a x86 host is referred as *emulation*. Most virtual machines have primitive support for debuggers, such as breakpoints, register and memory inspection, and code dumping.

Virtual machines can benefit to distributed debugging [4]. For example, debugging grid applications not only entails locating, analyzing and correcting errors, it also requires testing over various hardware and network configuration/combinations. While changing the topology of a production Grid is impossible, instantiating multiple virtual machines with different hardware/network configurations is far easier.

### 4.2. Virtual machines for execution replay

A virtual machine based approach has many benefits regarding execution replay. At the virtual machine level, the execution monitoring can be done at the assembly instruction granularity [3]. Hence, monitoring is independent of the programming language of the system. Moreover, a VM based approach makes no assumptions about the system architecture or specific design patterns. Hence, using a virtual machine approach saves the developer from defining the monitoring concern early in the design process and from manually adapting the system to a specific debugging tool.

A virtual machine approach avoids the irreproducibility effect, as it offers the debugger complete control of the "emulated" hardware. Hence, every piece of non determinism may be eliminated, for example interrupt arrival may be replayed at the exact same instruction as during the monitoring.

Nevertheless, a virtual machine approach does not avoid the probe effect: indeed virtual machines have an effect on temporal behaviors of systems that may be exploited in honeypot detection [5]. In Grid architectures, many administrators deploy virtual machines to be able to share nodes among multiple applications that runs on specific Unix distributions. Hence, we may argue that the release version of a system could run over a virtual machine, thus ignoring the probe effect.

The probe effect caused by the virtual machine can be turned in favor of debugging. Indeed, one of the difficulties with synchronization errors and race conditions is that they may arise only when certain conditions are met, *e.g.*, a pathological scheduling. Classical approaches of monitoring may only observe these situations, whereas a virtual machine may control the non determinism that gives rise to these conditions. For example, a developer could slow down or even stop one node in a distributed architecture to impose the winner of a race so as to observe the behavior of the rest of the system.

When a developer locates an error using execution replay, he might want to resume the execution in order to check that he has removed the error. But, a distributed system exchanges messages over network protocols. Hence, after an execution is resumed network connections may break because of timeouts. Thus, it might not be possible to check the error is corrected [9]. Because time is virtual, a virtual machine approach addresses this issue.

## 5. Implementation issues

Execution replay is done first by the monitoring phase that gathers all relevant information that is later used when replaying the execution. In this section, we describe how to make a virtual machine monitor, unwind and finally replay an execution. We also discuss how to synchronize multiple virtual machines to keep a distributed system coherent during backwards and replay execution. We focus on the Intel IA-32 architecture [6] and the QEMU processor emulator [2].

### 5.1. Monitoring, reverse and replay executions

QEMU uses dynamic code translation to emulate target operations in two steps: (*i*) QEMU fetches a translation unit, *i.e.*, a contiguous bloc of target instructions ending with a branch instruction or a virtual CPU state change such as an interrupt, (*ii*) the translation unit is translated into host code and executed.

To be able to replay an execution, we must first add monitoring to save information lost during the computation. We propose to store this information into a circular buffer. To illustrate this process, we use the example shown in Figure 1. In the example, target x86 instructions[2] are shown on the left side of the figure. The code fragment performs four operations. First the register `ebp` is incremented by one. Second, the `inb` instruction copies the value on the CPU port number `0x60` into the register `eax` before the latter is copied to the address pointed by `ebp`. Finally the code jumps to the address `next`.

In the example, the target code fragment is first executed with monitoring, second it is reverse executed and finally replayed. For each step, the translated code[3] is shown on the right side of the figure and is detailed in the rest of the section. The `cBuf` variable points the top of the circular buffer. The bottom part of the figure shows the evolution of the circular buffer during the execution of the translated code.

#### 5.1.1. Monitoring

During monitoring, we must save all information necessary for the reverse execution, *i.e.*, values that are overwritten or lost during the computation. Moreover, we must also collect information for the replay, *i.e.*, values that cannot be reproduced identically, such as a generated random number. Some instructions, such as `inc`, do not need extra computation because the old value of the operand may be directly deducted from its new value. On the contrary, the

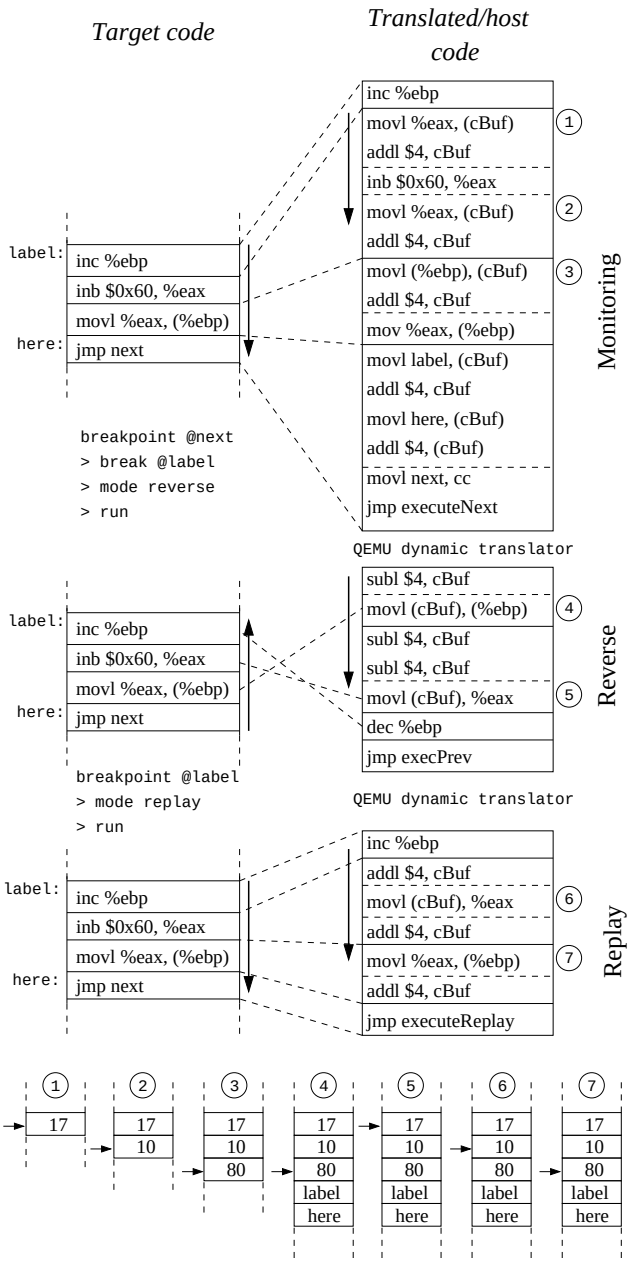


**Figure 1. An example target code first monitored, second reverse executed and finally replayed**

[2]The code that the developer wants to debug.

[3]The native code running on the real machine.

`inb` instruction overwrites the contents of the `eax` register. Hence, before the `inb` instruction is emulated, monitoring code "pushes" the value of `eax` (17) on top of the circular buffer. Here, saving the previous content of `eax` is not enough, indeed during the replay execution we want the value read on the port to be the same as during monitoring. Hence after the emulation of `inb`, the value read (10) on the port is also "pushed" on top of the circular buffer.[4]

During monitoring, it is necessary to save the path taken by the execution for the reverse execution mode to know how to backward execute. But the IA-32 opcodes are not backward readable. Moreover, a translation unit may end either on a branch instruction or because of an interrupt. Hence, when the execution reaches the end of the translation unit, *i.e.*, the `jmp` instruction, the monitoring code stores the address of both the beginning and the end of the translation unit. Finally, the target address of the jump is stored to allow QEMU to translate the next translation unit to be executed.

#### 5.1.2. Reverse execution

To make QEMU reverse execute the code fragment that was executed most recently, we must modify the dynamic translator to combine the semantics of the instructions to emulate with the information gathered during the monitoring process. First, the ordering of target instructions must be inverted in the translated code. Then, each target instruction must be translated into a sequence of instructions that undoes its effect. For example, the `movl` instruction is translated to restore the value referenced by `ebp` from the circular buffer (17). For the `inb` instruction, the `cBuf` pointer is updated to skip the saved value read on the CPU port during the monitoring (80). After that, the value of `eax` is restored from the circular buffer (10). Finally, the `inc` instruction is simply translated into a `dec` instruction.

#### 5.1.3. Execution replay

The execution replay mode is very similar to the emulation mode. Indeed, most instructions of the Intel IA-32 architecture have deterministic effects. For example, the `inc` instruction, as during the monitoring phase, is left unchanged. On the contrary, we do not want to re-execute the `inb` instruction, but we want it to return the same value as during the monitoring phase. Hence, the `inb` is translated into a `movl` instruction that fetches the value read originally and stores it into `eax` (10). Finally, the `movl` instruction is replayed identically.

During the replay phase, values that are overwritten during the computation are not saved a second time, but the `cBuf` pointer is nevertheless advanced identically as during the monitoring phase. This way it is possible to reverse execute and replay the same code fragment multiple times.

Reverse and replay executions allow to analyze the propagation of an error inside a node. Distributed errors typically involves interactions between multiple nodes of the distributed system. Hence, it is necessary to synchronize the reverse and replay executions of multiple virtual machines for the developer to locate the source of an error.

### 5.2. Synchronizing QEMUs

When an error is suspected, the developer stops the execution of all running QEMU instances. Because a developer typically examines one flow of control at a time, the developer might want to examine a single node of the distributed system, by reverse executing, and replaying it. Reverse executing one node in a distributed system puts the whole system in an incoherent state. For example, Figure 2 shows a distributed system with two nodes $A$ and $B$ that exchange messages over the network. Suppose the developer suspects an error and stops both nodes in state $s_3$. The developer backtracks the node $B$ in state $s_2$. If the developer now wishes to examine node $A$, the system is in an incoherent state because in state $s_3$ node $A$ has received a message from $B$ that has not been sent yet. Thus, when examining $A$, the emulator should backtrack $A$ at least to just before message $M_2$ has been received.

More generally, every time a node is reverse executed, if it undoes a message sent, the receiver should be reverse executed to just before the reception of that message. Similarly, when replaying execution on one node, if that node redoes a message reception, the sender should be executed forward just after the sending. This two rules applies recursively in order to keep the whole system coherent during reverse and replay execution.

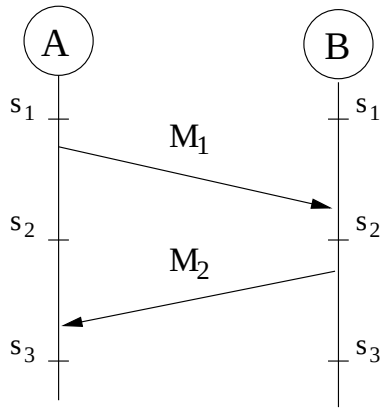


**Figure 2. Synchronizing QEMUs**

[4] As a note, the `inb` instruction has been left unchanged in the translated fragment. Normally, QEMU changes it into a sequence of instructions that emulates the CPU port. For the sake of clarity and because we do not modify that part of the translation, we left it as it in the example.

To achieve this overall synchronization, we need to modify the emulated network hardware in order to keep track of messages between nodes. Each node should mark messages sent with a unique identifier, because the send order is not always the same as the receive order. Then, when backwards or forwards executing a node, that node must notify the receiver or sender to update accordingly.

## 6. Conclusion

In this paper, we have proposed to design a replay execution debugger using a virtual machine approach. We have outlined a possible implementation of execution replay at the assembly instruction level using a reverse execution to backtrack one execution.

A virtual machine approach addresses the irreproducibility problem. While it does not avoid the probe effect, an implementation of a virtual machine may mimic a real machine well enough to capture distributed errors such as race conditions. Moreover, a virtual machine approach may more easily allow a developer to provoke synchronization errors, *i.e.*, by changing the winner of a race condition.

We choose to place our approach at the assembly language level. While this choice allows to build a generic execution replay debugger for distributed systems, there is a usability problem. Grid applications sometimes run over different platforms with multiple programming languages. Indeed, debugging Grid applications that use a mix of C, Fortran, and Java is almost impossible at the assembly level without source support. In the future, we plan to work on GDB support for distributed reverse and replay execution.

Classic debuggers features primitives, such as breakpoints and watch expressions, that poorly address distributed issues. In our future work, we will try to address usability issues. In particular, we plan to work on manipulating interactions inside a distributed system to allow a developer to manually provoke synchronization or race condition errors.

## References


[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, Oct. 2003. ACM Press.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, Anaheim, CA, USA, Apr. 2005. USENIX.

[3] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th international Symposium on Operating System Design and Implementation*, pages 211–224, Boston, MA, USA, Dec. 2002. USENIX.

[4] A. Ho, S. Hand, and T. L. Harris. Pdb: Pervasive debugging with Xen. In *Proceedings of the 5th International Workshop on Grid Computing*, pages 260–265, Pittsburgh, PA, USA, Nov. 2004. IEEE Computer Society.

[5] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Proceedings of the 6th IEEE Information Assurance Workshop*, pages 29–36, West Point, NY, USA, June 2005. IEEE Computer Society.

[6] Intel Corportation. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corportation, 2001.

[7] R. B. Kilgore and C. M. Chase. Re-execution of distributed programs to detect bugs bidden by racing. In *Proceedings of the 30th Annual Hawaii International Conference on System Sciences (HICSS)*, volume 1, pages 423–442, Maui, Hawaii, USA, Jan. 1997. IEEE Computer Society.

[8] P. A. Laplante. Heisenberg uncertainty. *ACM SIGSOFT Software Engineering Notes*, 15(5):21–22, Oct. 1990.

[9] M. Marwah, S. Mishra, and C. Fetzer. TCP server fault tolerance using connection migration to a backup server. In *Proceedings of 2003 International Conference on Dependable Systems and Networks*, pages 373–382. IEEE Computer Society, June 2003.

[10] 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 (PLDI)*, volume 31, pages 258–266, Philadelphia, PA, USA, May 1996. ACM Press.

[11] M. E. Shobaki and L. Lindh. A hardware and software monitor for high-level system-on-chip verification. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pages 56–61, San Jose, CA, USA, Mar. 2001. IEEE Computer Society.

[12] D. F. Snelling and G.-R. Hoffmann. A comparative study of libraries for parallel processing. *Parallel Computing*, 8(1–3):255–266, Oct. 1988.

[13] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMWare workstation's hosted virtual machine monitor. In *Proceedings of 2001 USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, June 2001. USENIX.

[14] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 265–272, Stockholm, Sweden, June 2000. IEEE Computer Society.

[15] J. J. Tsai, K.-Y. Fang, and H.-Y. Chen. A noninvasive architecture to monitor real-time distributed systems. *IEEE Transactions on Software Engineering*, 23(3):11–23, Mar. 1990.

[16] J. A. Wheeler and W. H. Zurek. *Quantum Theory and Measurement*. Plenum Press, Princeton, 1983.