Dealing with Reversibility of Shared Libraries in PDES

Davide Cingolani
Sapienza, University of Rome
Italy
cingolani@dis.uniroma1.it

Alessandro Pellegrini
Sapienza, University of Rome
Italy
pellegrini@dis.uniroma1.it

Markus Schordan
Lawrence Livermore National Laboratory
USA
schordan1@llnl.gov

Francesco Quaglia
Sapienza, University of Rome
Italy
quaglia@dis.uniroma1.it

David R. Jefferson
Lawrence Livermore National Laboratory
USA
drjefferson@llnl.gov

ABSTRACT

State recoverability is a crucial aspect of speculative Time Warp-based Parallel Discrete Event Simulation. In the literature, we can identify three major classes of techniques to support the correct restoration of a previous simulation state upon the execution of a rollback operation: state checkpointing/restore, manual reverse computation and automatic reverse computation. The latter class has been recently supported by relying either on binary code instrumentation or on source-to-source code transformation. Nevertheless, both solutions are not intrinsically meant to support a reversible execution of third-party shared libraries, which can be pretty useful when implementing complex simulation models.

In this paper, we present an architectural solution (realized as a static C library) which allows to transparently instrument (at runtime) any third party shared library, with no need for any modification to the model's code. We also present preliminary experimental results, based on the integration of our library with ROSS and ROOT-Sim simulators.

CCS CONCEPTS

•**Computing methodologies** →**Discrete-event simulation;** *Simulation environments; Simulation tools;* •**Hardware** →*Reversible logic;* •**Software and its engineering** →Software libraries and repositories;

KEYWORDS

PDES, Speculative Processing, Code Instrumentation, Reversibility

ACM Reference format:
Davide Cingolani, Alessandro Pellegrini, Markus Schordan, Francesco Quaglia, and David R. Jefferson. 2017. Dealing with Reversibility of Shared Libraries in PDES. In *Proceedings of ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation, Singapore, May 2017 (PADS'17),* 9 pages.
DOI: 10.475/123_4

PADS'17, Singapore

DOI: 10.475/123_4

1 INTRODUCTION

In the context of Parallel Discrete Event Simulation (PDES) [15], among the various synchronization protocols, the speculative Time Warp [18] one has been proven to be particularly effective, as it is relatively independent (in terms of its run-time dynamics) of both the the simulation model's lookahead and the communication latency for exchanging data across threads/processes involved in the simulation platform. This allows Time Warp systems to guarantee a high performance as well in systems that are not tightly coupled and/or encompass millions of processors [3].

The speculative nature of Time Warp allows simulation events to be processed at any LP independently of their safety (or causal consistency). If an event is a-posteriori detected to be violating causality, its effects on the simulation state are undone, via the *rollback operation*. Correctly and efficiently rolling back the simulation state is therefore a fundamental building block for an effective optimistic simulation platform.

Among the different approaches proposed in the literature to rollback the simulation state, the main two families which are considered are *checkpoint-based* [18] and *reverse computing-based* [7], depending on the algorithmic technique which is used to bring one simulation state to a previous (consistent) snapshot. The checkpoint-based rollback operation has been thoroughly studied to reduce its cost (both in terms of memory and CPU usage), either by reducing the checkpointing frequency (the so-called *sparse* or *periodic state saving*) [4, 13, 21, 22, 27, 29, 33] or by reducing the amount of data copied into a state snapshot (the so-called *incremental state saving*) [24, 36].

The reverse computing-based rollback operation, which tries to cancel the non-negligible memory footprint produced by the state saving technique, relies on *reverse events*, which can be generated either manually [7] or automatically [8, 19, 30, 31]. With respect to automatic generation of reverse events, the various proposals address it by relying either on binary instrumentation [8, 19] or on source-to-source transformation [30, 31]. Nevertheless, none of these solutions is able to deal with third-party shared libraries, which could be regarded as an important building block for the development of complex simulation models. To mention some, libraries such as ALGLIB [32], GSL [16], FFTW [14], LAPACK [11], or BLAS [20] might be necessary for the description of statistical

or algebraic processes, proper of a large number of simulation scenarios.

These third-party shared libraries are not *optimism-aware*. In fact, they are devised for application scenarios which always operate on *committed* data, which is something that speculative synchronization protocols, such as Time Warp, intentionally do not provide continuously. While static binary instrumentation and source-to-source transformation could be directly used on these shared libraries to make them *reversible*, their applicability might fall either due to the lack of source code (in the case of closed-source libraries) or due to the fact that instrumenting shared objects could produce system-wide effects to other programs not related to optimistic simulation.

In this paper, we propose an alternative technique, based on the concept of *lazy instrumentation*, to complement static and source-to-source instrumentation, for x86 systems. This technique allows to intercept any call to any third-party shared library function, allowing to create (transparently to the simulation model developer) an instrumented version which could be easily coupled with any reversible simulation engine. Moreover, our technique allows to quickly switch between instrumented and non-instrumented (original) functions, opening to the possibility of fine-grained runtime self-optimization of the simulation run (similarly to the technique proposed in [25]) and to a different behavior of the engine when dealing with model vs. platform code.

Our technique allows to be transparently embedded into any Time Warp-based simulation engine, equipped with a reversible memory management module, via a set of API functions which allow to tune its functioning at simulation startup. To assess the viability of our proposal, and to illustrate as well the working simplicity, we present preliminar experimental results collected using two simulation engines, namely the Rensselaer's Optimistic Simulation System (ROSS) [6], and The ROme OpTimistic Simulator (ROOT-Sim) [23].

The remainder of this paper is structured as follows. In Section 2 we discuss related work. Section 3 presents the design choices below our proposal, and its implementation. Experimental data to assess the viability and effectiveness of our proposal are finally reported in Section 4.

2 RELATED WORK

Despite the fact that reversibility grounds its roots in the 1970's [37], to the best of our knowledge no one has explicitly targeted instrumentation of third-party shared libraries for software reversibility purposes, in any computer science application.

The idea of supporting the rollback operation in the context of Time Warp systems by relying on reverse computation rather than on snapshot restoration dates back to 1999 [7], but at the time the reverse code was hand-generated. A first attempt to automatically generate reversibility code can be found in [19], where control flow analysis is used to generate code which allows to reconstruct the execution path taken in the forward code. Differently to our proposal, reverse code is generated at compile time, preventing the possibility to operate to any number of third-party libraries.

Another recent work, which is evaluated in this paper in conjunction with our proposal, is in [30]. In this work, the authors perform source-to-source transformation of C++ code based on the ROSE compiler infrastructure [28], intercepting all memory modifying operations and recording information about the performed updates in a data structure that is used to reverse the effects of memory updates. As mentioned, our proposal specifically targets all these scenarios where source code of third-party libraries is not available, thus preventing source-to-source transformation from being a viable solution.

The works in [24, 36], similarly to what we do, rely on static binary instrumentation to track memory updates during the forward execution of the events. Nevertheless, their goal is to use this information to generate periodic incremental checkpoints, which we avoid by design in our proposal.

Static binary instrumentation is used in [8] to generate so-called *undo code blocks*, which are packed data structures which keep machine instructions generated on the fly to undo the effects of the forward execution of events. While this is a proposal similar in spirit to this work, the authors in [8] do not account for the presence of third-party libraries, therefore limiting the degree of programmability of simulation models.

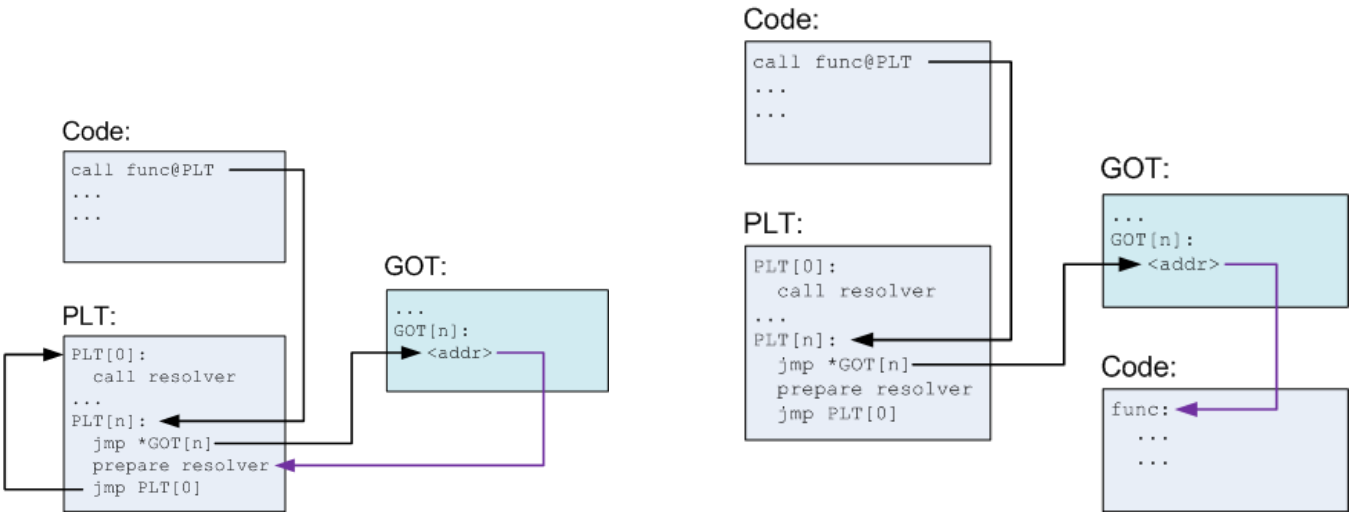
Our proposal is also related to a number of works in the field of program execution tracing (see, e.g., [1, 2, 26, 38]) for debugging, vulnerability assessment and repeatability. These approaches provide detailed analysis of changes in the state of the program, and of the execution flow. Nevertheless, these works do not explicitly deal with the possibility to reverse a portion of the program's execution by relying on runtime-generated reverse instructions.

The US patent in [17] explicitly deals with reversibility of shared libraries within executables. Yet, differently from our proposal, the goal of this work is to make reversible the linking process, thus allowing for different versions of the library to be attached to the same program. Differently, we are interested in undoing the effects of shared libraries on the memory map, to support a reversibility-based rollback operation.

3 REVERSIBILITY OF SHARED LIBRARIES

Before discussing the approach that we undertake to enact software reversibility of generic third-party shared libraries, let us summarize how third-party libraries interact with an executable, taking as an example Linux systems relying on the Executable and Linkable Format (ELF). Whenever the compiler determines that some function referenced in the source belongs to a shared library, it introduces in the program's image additional pieces of information to let the system, at runtime, *resolve* any reference to that some function to its actual implementation. In particular, the compiler:

- Registers the name of the shared library in the program's image. This name often comes with the actual version of the library in it, so that if the executable is moved to a different environment where the correct library's version is not present, the loader fails to resolve any call, to avoid undefined behaviors;
- Reserves an entry in a special table, called the *Procedure Linkage Table* (PLT) is reserved for the specific library function. Any call to that function will actually refer the associated PLT's entry, which keeps enough space to host a couple of machine instructions;



(a) GOT and PLT, as organized by the compiler, before any runtime resolution is carried out. PLT points to the corresponding GOT entry, which in turn points to the same PLT entry. This circular reference allows the activation of the resolver, stored in `PLT[0]`, in order to determine where the symbol is actually placed in memory.

(b) GOT and PLT organization after the first call to a library function is issued. The dynamic linker has resolved the virtual address of the function, which is stored into the corresponding entry in the GOT table. Any other invocation to the function will not activate the linker, rather the address of the function will be immediately available for activation.

Figure 1: Resolver, GOT, and PLT hooking.

- For any entry in the PLT, it reserves the corresponding entry in the *Global Offset Table* (GOT), which only stores a memory pointer.

The need for two tables arises due to the *lazy binding* policy adopted by the dynamic loader. In fact, PLT and GOT reference each other in a way that allows the system to know whether a library function is being called for the first time or not. In the positive case, the library symbol is resolved, otherwise the (already-resolved) function is simply called.

To illustrate this mechanism—this is exactly where our proposal acts so as to generate reversibility-enabled copies of third-party library functions—let us suppose that a program relies on the shared library's function `func`. The GOT and PLT tables are organized as in Figure 1. As mentioned, the call to `func` is actually a call to an entry of the PLT table, namely `PLT[n]`, where `n` is the entry associated with `func`. The first entry of this table, namely `PLT[0]`, is a special entry, which keeps an instruction to call the `resolver`, i.e. the function of the dynamic linker which is in charge of determining where the entry point of any library function is in memory. Once the first call to `func` is issued, the code in `PLT[n]` takes control. By PLT contruction, the code jumps to the address pointed by the corresponding `GOT[n]` entry. At program startup, this address points to `PLT[n]` itself, specifically to a code snippet which prepares (on the stack) the parameters needed by the dynamic linker to determine what library call caused its invocation. Then, the actual resolver is called, by jumping to `PLT[0]`. The resolver performs the resolution of the actual address of `func`, places its actual address into `GOT[n]` and calls `func`. The GOT/PLT organization, after the symbol's resolution, is depicted in Figure 1(b). Any other call to `func` will not cause the activation of the dynamic resolver, as the address stored in `GOT[n]` now points to the actual virtual address of `func`.

In order to instrument third-party library function calls, we specifically intercept the above-described mechanism. In particular, our approach relies on a static library to be linked to the executable, which we refer to as `libreverse`[1]. This library contains a *program constructor*, namely a function which is activated by the program loader before giving control to the actual `main` program. The goal of this constructor is to replace the call to `resolver` to a different function, exposed by the library itself, which alter the behavior of the latter part of the dynamic linking process. In particular, the custom resolver takes the following steps:

(1) Similarly to the dynamic linker's resolver, it determines what is `func`'s entry point virtual address;
(2) Once `func`'s address is identified, it creates a copy of the whole function in memory, instrumenting any instruction which has a memory operand as the destination—namely, a *memory-write* instruction;
(3) The instrumentation is carried in a way such that before executing the actual memory write operation, control is given to a *trampoline* which activates a reversibility-oriented facility of `libreverse`;
(4) An entry in a custom table, called *Library Activation Trampoline* (LAT), is reserved. This entry keeps a small portion of code to determine whether the instrumented version of the library should be called or not;

[1]The whole source code of the library is available at https://github.com/HPDCS/libreverse.

(5) The address of LAT[n] is stored into GOT[n], allowing any future activation of func to directly give control to the code in LAT[n];
(6) Control is given to LAT[n], in order to perform the actual function call.

This general scheme allows to intercept *any* call to *any* function in *any* third-party shared library, therefore making them aware of the reversibility requirements required by Time Warp-based simulations, necessarily to support the rollback operation. All these points above require special care, and we therefore describe in the following how these are supported by libreverse.

3.1 Intercepting the Dynamic Linker's Resolver

Although the steps taken by the dynamic linker's resolver are mostly standardized, there could be some variability across systems and versions of the linker with respects to the actual steps taken. To make libreverse of general availability, we want our custom resolver to take the same steps as the system's dynamic linker. To this end, we take the following steps to ensure portability across systems and linker versions.

As mentioned, when the program is launched, libreverse's constructor is activated, which replaced in PLT[0] the address of the resolver with a custom one. Nevertheless, since this custom one should be compliant with the system's one, libreverse does not contain the custom resolver's code, rather it generates it at program startup. In particular, any dynamic linker's resolver has to perform these tasks:

- Determine whether the image of the shared library is mapped into the program's image, if not it has to be mmap'ed;
- Determine where is the function's entry point in the library image. This is commonly done by relying on a fast hash-function based mechanism, relying on the data stored in the .dynsym section in case of an ELF executable;
- The address is stored in GOT[n], where n is passed as an argument on stack;
- The function is activated directly by the resolver.

The last point is where we hook our custom code. In particular, the activation of the library function is made by relying on an *indirect jump*. On x86 systems, this is implemented by an instruction in the form jmp *%reg, i.e. the address of the target function is stored into a register, which is used as the destination of the jump instruction. Once libreverse's constructor takes control, it creates a copy of the system resolver's code, and starts scanning its bytes until such an instruction is found. This jump is then replaced with a *direct jump*, whose target is a function within libreverse which takes care of instrumenting the target function, before the control is given to it.

This strategy allows libreverse to attach itself to any version of the dynamic loader's resolver, independently of the actual way the identification of the function's symbol within the shared library's image is carried out.

3.2 Instrumentation of Library Functions

Once a library function is first called, by the above interception of the dynamic linker's resolver by libreverse, we are able to take control right after the address of the function is identified. At this point, in order to perform the actual instrumentation of the function, we must determine its size. To this end, we recall that the executable keeps track of the library file on disk storage. We are therefore able to navigate the path of the library file, and open it. A shared library on Linux systems is represented as an ELF file. By inspecting the symbol's table of this file, we can determine what is the actual size of the called function.

At this point, the instrumentation process can take place. We allocate a memory area of the same size as the original function via an mmap, making it both writable and executable, and we copy the whole content of the functions' binary representation in it. This will be the working copy of the function, which we can inspect and alter accordingly, in order to enable reversibility of its actions.

The instrumentation process requires two *logical steps*. The first one entails determining the total number of assembly instructions which compose the function. Among them, we then must identify all memory-write instructions, and properly alter them in order to generate on the fly *reverse instructions*, namely assembly instructions whose execution undoes the effects of the original instructions in memory. To this end, we must determine both the destination address of the memory write instruction, and its size.

We note that these two steps require two different levels of detail (and, consequently, of complexity). Indeed, to determine the number of instructions which compose the library, we do not need to get into the *semantics* of the instructions themselves, which is a non-minimal optimization given that our target is the x86 architecture. In fact, the x86 ISA is a variable-size one. This means that the length in bytes of a single assembly instruction cannot be determined beforehand. Only by interpreting the opcode it is possible to determine the exact amount of parameters to the instruction, and therefore its length.

For the sake of performance, libreverse is equipped with two different disassemblers. The first one, which we call a *length disassembler*, is a fast table-based routine which only tells what is the length of the actual instruction in bytes, and gives a reference to the actual opcode[2]. The second disassembler which is included into libreverse is a *full disassembler*: it fully decodes the bytecode representation of the instruction, evaluating all its fields, allowing to extract the data of interest. The execution of the length disassembler is 3 times faster than the full disassembler, on any x86 instruction (i.e., independently of its length).

Therefore, libreverse enacts the instrumentation process in the following way. The length disassembler is invoked on the initial address of the function, returning the size of the first instruction and a pointer on its actual opcode. This opcode is matched to a table which tells whether the instruction *could* entail a memory-write operation. In the negative case, the next instruction can be identified by inspecting the bytecode located n bytes after the initial address, where n is the length returned by the length disassembler. In the positive case, the full disassembler is invoked on the same memory location. This allows to determine whether the involved instruction is *actually* a memory-write one and, if it is so, it allows to extract the size of the memory write (which in case of a simple mov instruction

[2] In fact, x86 instructions can be preceded by an arbitrary number of *prefixes*, so that the first byte in a given instruction is not necessarily the opcode.

```
save CPU context (except RIP)
call reverse
restore CPU context (except RIP)
<original instruction>
jmp <address>
```

Figure 2: The instruction trampoline

is encoded in the binary representation of the instruction itself) and the destination address of the memory-write operation.

At this point, the instrumentation process *replaces* the memory-write instruction with a *jump* to a code snippet (which we call the *instruction trampoline*) generated on the fly. This snippet is placed into an additional table, called the `INSTRUCTIONS` table. This is a table which, for each memory-write instruction, keeps a portion of code which prepares the required information to generate the associated reversible instruction. Since the number of memory-write instructions is not known beforehand, the `INSTRUCTIONS` table is pre-allocated keeping the space for a certain number of trampolines. If the space in the `INSTRUCTIONS` table is exhausted, a new table is silently allocated.

The instruction trampoline's code is organized as in Figure 2. The first required action is to save the CPU context. This is because the original library function must be unaware of the execution of all the injected code. Unfortunately, since the code was placed after the program's compilation, standard `setjmp/longjmp` functions cannot be used, as we are explicitly breaking System V ABI's calling conventions and *caller save* registers are not saved by the code. Therefore, our solution is to perform a fast CPU-context save by pushing all required general-purpose registers and the flags register. Since the code is crafted directly in assembly language, we use only *callee-save* registers, and we push all of them on stack. In this way, we do not need to concern about registers used by functions called by the trampoline, as their code is compiler-generated, and therefore respects the calling conventions. Since we have saved all callee-save registers, the consistency of the program's execution is preserved.

After having saved the CPU context, we issue a call to `reverse`, a `libreverse` internal function which computes the target memory-write address and generates the corresponding reverse instruction. According to the addressing mode of the x86 architecture, each memory address is identified by the expression $base\ address + (index * scale) + displacement$. While the parameters *scale* and *displacement* are already encoded in the instruction binary representation, *base address* and *index* refer to the content of registers, which can be evaluated only at runtime. Therefore, once the control is given to the trampoline of a certain instructions, some data to allow the computation of the memory-write target address (and the size of the write, when available) are placed on stack. These data are the outcome of the instrumentation process, and are organized as in the following structure:

```
struct insn_entry {
        char flags;
        char base;
        char idx;
        char scale;
        int  size;
        long long offset;
}
```

where `flags` tells which are the relevant fields to recompute the target address, or to identify the class of data-movement instructions, as we will explain later in details; `base` keeps the (3 or 4 bits) base register binary representation; `idx` keeps the (3 or 4 bits) index register binary representation; `scale` is used to store the scale factor of the addressing mode; `size` holds the size (in bytes) of the memory area being affected by the memory-write instruction (when available at disassemble time); `offset` keeps the displacement of the addressing mode[3].

By relying on this information, the `reverse` function can determine the size and the target address of the memory-write instruction. This information is used to generate the corresponding reverse instruction, as we will discuss later. Nevertheless, so far, the original instruction has not been executed yet. As mentioned, the original instruction's bytecode is replaced with a jump to the corresponding entry in the `INSTRUCTIONS` table. In order to execute the original instruction, we copy the binary representation of the instruction directly within the corresponding `INSTRUCTIONS`' entry, after the call to the `reverse` function. Nevertheless, the original instruction might require contextual information in order to execute properly. This is due to the fact that many instructions in the x86 ISA use *relative* references. As an example, consider an operation used to store a value into a *local* variable. These variables are stored on the stack, and are often referred using a displacement from either the base frame pointer, or from the stack pointer. Therefore, before giving control to the copy of the original memory-write instruction withing the `INSTRUCTIONS` entry, we restore the CPU context (except for the value of the `RIP` register, the program counter). This allows to correctly execute a large set of instruction, although we must explicitly account for the fact that the value kept by `RIP` is different from the original execution context.

This latter point deserves an additional discussion. In fact, in the 64-bit version of the x64 architecture, a special addressing mode, which is called `RIP`-relative, allows to target symbols (e.g., variables) encoding in the instruction a displacement to the current value of the `RIP` register. This addressing mode is particularly important for library functions. Indeed, a shared library can be remapped to any virtual memory address range, depending on the set of libraries and/or runtime dynamics. Therefore, to reduce the overhead related to library loading, shared libraries code is generated by compilers as *position-independent code* (PIC). A PIC library has no indirect reference to any library or variable. This means that *any* reference within a library is expressed as a displacement with respect to the current instruction. In the 64-bit x86 ISA, this entails a huge usage of the `RIP`-relative addressing mode.

[3]We provide 64-bits space in the `insn_entry` structure due to the fact that the `x86_64` assembly language allows one single instruction, namely `movabs`, to directly use a 64-bits addressing mode. In all the other cases, only 32 bits of the `offset` field are actually used.

```
   mov    %fs:platform_mode@tpoff,%eax
   cmpb   $0x0,%eax
   jz     1f
   call   original_function
   ret
1: call   instrumented_function
   ret
```

Figure 3: Entry of the LAT table (x86 64-bit version)

To cope with this issue, we cannot simply restore the whole CPU context, including the value of RIP. In fact, at the original address we no longer have the original instruction. To execute its copy, RIP *must* point to the copy, which is at a different address. Therefore, to correctly execute memory-write operations which rely on RIP-relative addressing, the only option is to *fix* the displacement. To this end, we rely on the length disassembler. In particular, this disassembler sets a gloabal (per-thread) flag whenever it encounters an assembly instruction which is using the RIP-relative addressing mode. Once such an instruction is found, the full disassembler is invoked on it, allowing to determine whether this addressing mode is used in the source or in the destination operand. In both cases, the offset is corrected. This correction is trivial: we can at any time determine what is the *additional offset* (either positive or negative) introduced by the fact that the instruction is being moved to a different location. Anyhow, the correction of RIP-relative addressing cannot be limited to instructions copied into the INSTRUCTIONS table. In fact, since we create a whole copy of the original library function, all RIP-relative addressing must be corrected. Anyhow, by relying on the above-described scheme, the correction can be actuated in place on the copy of the instructions.

To complete the instrumentation process of the library function, we iterate over all the instructions carrying on the aforementioned steps, until we reach the end of the function. As mentioned, we can identify the end of the function by inspecting the library's ELF symbol table, to determine its total length in bytes. The final instruction of the instruction trampoline must give control back to the library function. Since there is a one-to-one mapping between the memory-write instruction and the entry in the INSTRUCTIONS table which keeps its instruction trampoline, the return to the function's flow can be implemented with a direct jmp instruction to the correct address.

After that the whole function is instrumented, we have to hook the altered version to the GOT/PLT invocation mechanism. As hinted before, we want to give the possibility to activate both the original version and the instrumented one, depending on the execution context. In particular, the reversibility facilities are only related to the execution of the simulation models' event handlers, while when executing in *platform mode* (i.e., when the control is taken by the simulation engine) we do not need to generate reverse instructions. In this latter case, for the sake of performance, we want to rely on the original version of the library functions, if they are used. To allow a fast switching between the two versions, we rely on the aforementioned LAT table. In particular, the *n*-th entry in the LAT, which corresponds to the current function being instrumented, is organized (for the case of 64-bit x86 Linux systems)

Figure 4: The index of assembly instructions built while instrumenting a library function.

as in Figure 3 (for a total of 24 bytes). The goal of this code is to check a (per-thread) global variable called platform_mode which tells whether a library function is invoked from the simulation engine level or from the application-level code. In the former case, the original (non-instrumented) library function is activated, while in the latter the instrumented version is called. To change the execution mode, libreverse offers an internal API function, named platform_mode(bool) which tells whether control is being passed to an event handler or it is returning from such a handler. It is the responsibility of the simulation engine, when integrating with libreverse, to properly use this function. Overall, the integration with the GOT/PLT invocation mechanism is simply done by placing, after the LAT entry is properly crafted, its address within the corresponding entry of the GOT table.

There are two classes of instructions which cannot be directly dealt with according to the aforementioned instrumentation scheme, rather require special management. One is the cmov instruction, which is managed directly in the trampoline. Specifically, in case of a cmov, we use 4 bits of the flags field to record what is the check to be emulated. The trampoline checks whether the bits are different from zero, and in the positive case the corresponding status bits are checked to determine whether the condition is met or not. Nevertheless, the values of status bits might have been already altered during the execution of the previous injected operations. To this end, the trampoline's code looks on the application stack for the old value, as stored during the CPU-context save phase. If the condition is met, the cmov is managed exactly like a standard mov.

The second one is the movs instruction, for which we use one bit of the flags field to let the trampoline know whether its invocation is related to such an instruction. In this specific case, the size flag tells only the size of one single iteration of the movs instruction. Therefore, to compute the total size, the trampoline's code checks the value of the rcx register, and multiplies it by size. The starting address of the write is then computed by first checking the *direction flag* of the flags register. In case this flag is cleared, the destination starting address is already present in the rdi register. If the flag is set, then the movs instruction will make a backwards copy, and therefore the (logical) initial address of the move is computed as rdi - rcx * size.

An additional note must be discussed to complete our overview of the instrumentation mechanism. In fact, in order to link the place where a library function has a memory-write instruction with the corresponding INSTRUCTIONS entry, a jmp instruction is used to replace the actual mov. Nevertheless, the x86 ISA has a variable length. If the size of the jmp (which is 5 bytes) is smaller than the size of the actual intercepted memory-update instruction, the remaining space can be easily filled using a nop. On the other hand, the memory-write instruction's representation could be shorted than 5 bytes—the classical example is the aforementioned movs, which is only 1-byte long.

In this case, libreverse "makes room" for the jump instruction, by coalescing multiple consecutive instructions within the same INSTRUCTIONS entry. This is done by continuing the disassembly of the library function, until enough room for the jmp is found. Nevertheless, there could be the possibility that the end of the function is reached before finding enough room. In this case, libreverse "backtracks" its execution by coalescing instructions *before* the memory-write instruction, until enough space is found. Anyhow, since the length of an assembly instruction is variable, it could be resource intensive to perform this latter action. To this end, while performing the forward instrumentation, libreverse builds an *instruction index*, as depicted in Figure 4. This index keeps, for each instruction, its size in bytes. Therefore, in case while coalescing instructions the end of the function is reached, it is possible to increase the size up to the required amount of 5 bytes by simply inspecting this index. Since the number of instructions that compose the function is not known beforehand, the instruction index is implemented as a wait-free resizable array, as described in [10].

While this approach solves the problem related to the needed amount of bytes to insert the jmp to the INSTRUCTIONS' table entry, it might pose an additional problem. Let us discuss the following example:

```
    jmp 1f
    movl $0x0, %eax
    movsl
1:  leave
    ret
```

In this case, the instrumentation process will detect that the movs is a memory-write instrunction, an will trigger the replacement with a jmp. Since movs is only 1-byte long, the coalescing procedure will try to expand over subsequent instructions. The next is the 1-byte long leave, so the coalescing procedure continues, until the end of the function is reached. At this point, since the total amount of bytes found amounts to three, the coalescing procedure inspects the instructions' index to determine how many instructions behind the movs should be taken to make enough room to the jmp. Since the movl $0x0, %eax is 5-byte long, the coalescing procedure takes it and halts. This give a grand total of 8 bytes (with respect to the 5 needed) to place the jmp. Nevertheless, this action will completely break the functioning of the program. In fact, the initial instruction in the example is a jmp which targets one of the instructions which will be moved into the INSTRUCTIONS' table entry, having the jmp target the middle of the (newly-inserted) assembly instruction.

To overcome this issue, we extend the aforementioned instructions index . In particular, the opcode retrieved by the length disassembler is matched against a second table, which tells whether the instruction could have as a parameter a reference to a different instruction (e.g., the case for a jmp instruction). In the positive case, the instruction index keeps a reference to the instruction. In case it is a reference to a future instruction, we keep track of this by relying on a fast hash table. Once the target instruction is reached, the link between the two is completed.

Whenever an instruction is moved to an entry of the INSTRUCTIONS table, the corresponding entry in the instructions index is flagged. At the end of the intrumentation process, a fast scan of the instructions index is performed, so as to determine whether some instruction referenced by, e.g., a jmp instruction has been moved into an entry of the INSTRUCTIONS table. In the positive case, the referencing instruction's offset is corrected, simply applying the corresponding shift to the displacement.

The instructions index becomes handy to solve a couple of additional issues, related to the way shared libraries are built. In particular, any library function within a library can reference any other function within the library itself. Since a call instruction, to invoke another function in the library, uses an offset in a way perfectly similar to a jmp, but this reference will not be found during the instrumentation process of the current function. Here, two situations might arise:

(1) *The function is exposed to the application and is actually used*: in this case, an entry in the PLT is present. This can be verified by inspecting the ELF symbol table of the running application. The call, therefore, is redirected to the corresponding PLT entry. While this might reduce a bit any optimization internal to the library, allows to perform a lazily instrumentation according to the scheme that we are presenting in this paper;
(2) *The function is exposed to the application but is not used by the program, or is an internal one*: in this second case, we cannot rely on the PLT to carry on the lazy instrumentation. We rather keep within libreverse a list of functions internal to this library which have already been instrumented due to this specific scenario taking place. If the target function is present in this list, then the call is redirected to this already-instrumented symbol. If it is not, then the target function is instrumented (exactly according to the whole aforementioned scheme) and then the symbol is added to libreverse's internal list.

As a last note, since libraries are often implemented with high performance in mind, nothing prevents to "break" the common idea of function—this is something that, e.g., happens extensively in glibc. In particular, one function might jump into the middle of another one, just to execute a portion of its code in case some optimized condition of the host system is detected. While this scenario can be detected in a way similar to calls to different library functions (i.e., the reference of the jmp is not resolved while scanning the function), handling this condition is less trivial, as it would entail some code flow analysis like the ones presented in [12]. Since such an analysis is out of the scope of this paper, and considering that glibc shows this behavior only in a handful of functions (like, e.g., memmove()), for the sake of simplicity we have simply replaced these functions with less-optimized ones which are statically linked to the executable.

3.3 Generation and Management of Reverse Instructions

The instrumentation architecture described so far allows, at runtime, to activate the reverse function just before any memory-update operation is performed. At this point, libreverse is notified of the application code's will to update the simulation model state, and therefore reverse instructions (to restore the state in case of a rollback operation) can be built on-the-fly.

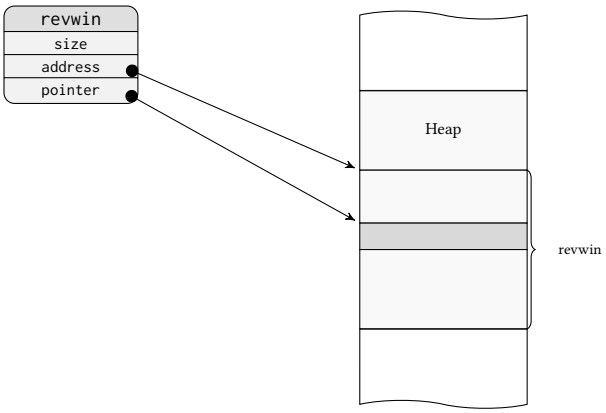


Figure 5: Revwin descriptor

If the activation of `reverse` is related to the execution (in the forward event) of a `mov` or a `cmov` instruction, the reverse instruction is built by accessing memory at the computed address and by reading the original value (i.e., the one before the write operation is executed). This value is placed within a data movement instruction as the source (immediate) operand, having as the destination address the same address. On the other hand, if the activation of `reverse` is due to a `movs` instruction, this can be easily determined by the size of the memory-write operation, as it is higher than the largest representable immediate[4]. The reverse instruction in this case can only be another `movs` instruction, having as the source operand a properly-allocated memory buffer where the original content has been copied upon reverse instruction generation.

The generation of reverse instructions is not a costly operation—except for the `movs` case where a memory buffer must be explicitly copied. Indeed, the set of instructions to be generated is very limited, and the opcodes are known beforehand. Therefore, we rely on a pre-compiled tables of instructions in which only the memory address and the old immediate should be packed within. With this approach, we pay an instrumentation overhead similar to that of incremental state saving solutions (see, e.g., [24]), but we are completely avoiding any generation of metadata, thus reducing the overhead for the installation of a previous snapshot during the execution of a rollback.

In order to allow the PDES engine to correctly interact with the management of reversibility of library function calls, `libreverse` intrinsically works with the notion of *events*. Since an event is an *atomic unit of work*, `libreverse` organizes reverse instructions in atomic blocks, on a per-thread fashion. In particular, in a way similar to the work presented in [8], reverse instructions are packed into the *reverse window* structure, which is depicted in Figure 5. Every time that a new reverse instruction is generated, it is inserted right before the address of `pointer`, whose value is then updated accordingly, so that they appear in reverse order with respect to the forward execution. This is a fundamental prerequisite to undo the effects of library calls within an event, as they can be undone by simply issuing a `call` to the first instruction in the reverse window (i.e., the one pointed by `pointer`). For a thorough description of the way reverse windows are managed, we refer the reader to our previous work in [8].

To let the simulation engine keep control of generated reverse instructions on a per-event basis, `libreverse` exposes an API function named `finalize_event()` which, upon invocation, allocates (again, on a per-thread basis) a new reverse window and returns a pointer to the currently-used reverse window. The simulation engine can then link this structure to any representation of the event in the just-descheduled LP message queue. To facilitate the management of the operations, `libreverse` offers two additional API functions, namely `execute_revwin()` and `cleanup_revwin()`. The former allows to execute the reverse instructions kept by a reverse window by giving control to the instruction pointed by `pointer`. The latter can be used to release all memory buffers related to a reverse window in case, e.g., an even is deemed committed or it is removed from the message queue due to the reception of an antimessage. By relying on this set of API functions, any Time Warp-based simulation engine can be easily integrated with `libreverse`.

3.4 Dealing with Memory Allocations and Deallocations

The last aspect to be dealt with to support a correct restoration of a previous state is related to the management of allocation/deallocation operations. In particular, if during the execution of a forward event the model's code invokes a library function which allocates memory, this memory logically belongs to the LP which is currently scheduled. While designing `libreverse`, we have assumed that the simulation engine has a per-LP memory map manager, such as the one in [25], as providing a memory manager is out of the scope of our approach. Therefore, in order to correctly link `libreverse` and the simulation engine, we must provide a means to map a forward memory allocation with the corresponding memory deallocation and vice versa.

To this end, `libreverse` offers two additional API functions, namely `register_alloc()` and `register_dealloc()`. These functions accept a function pointer each, which are defined as `void *(*allocate)(void *ptr, size_t size)` for the former function, and `void (*deallocate)(void *ptr)` for the latter. These pointers allow to bridge the internals of `libreverse` with the simulation engine's memory manager, so that whenever a library allocates some memory a call to the `deallocate()` function is placed within the reverse window, while when a chunk of memory is deallocated, a call to `allocate()` is similarly stored. We emphasize that having the `allocate()` function accept a pointer is a strategic choice to allow piece-wise-deterministic replay of events upon a rollback operations, allowing to retrieve buffers at the same virtual addresses, and therefore support a memory map laid out in a generic way.

4 EXPERIMENTAL RESULTS

5 CONCLUSIONS

REFERENCES

[1] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/. (????).
[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. *SIGPLAN Notices* 35, 5 (2000), 1–12.

[4] We note that, by using this approach, a possible `movs` instruction involving few bytes of memory is negated using a standard `mov` instruction, which is nevertheless correct, and possibly more efficient.

DOI: http://dx.doi.org/10.1145/358438.349303

[3] Peter D. Barnes, Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. 2013. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13*. 327. DOI: http://dx.doi.org/10.1145/2486092.2486134

[4] Steven Bellenot. 1992. State skipping performance with the Time Warp operating system.. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS)*. 53–64.

[5] RR Branco. 2007. Ltrace internals. *Linux Symposium* (2007). http://ols.fedoraproject.org/OLS/Reprints-2007/OLS2007-Proceedings-V1.pdf

[6] Christopher D Carothers, David W Bauer, and S Pearce. 2000. ROSS: a High Performance Modular Time Warp System. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 53–60.

[7] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation* 9, 3 (1999), 224–253.

[8] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2015. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*. ACM Press.

[9] F.b Cortellessa V.a Quaglia. 2001. A checkpointing-recovery scheme for Time Warp parallel simulation. *Parallel Comput.* 27, 9 (2001), 1227–1252. http://www.scopus.com/inward/record.url?eid=2-s2.0-0035427584

[10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2006. Lock-Free Dynamically Resizable Arrays. In *Proceedings of the 10th international conference on Principles of Distributed Systems*. Springer-Verlag, 142–156. DOI: http://dx.doi.org/10.1007/11945529_11

[11] James Demmel. 1991. LAPACK: A portable linear algebra library for high-performance computers. *Concurrency: Practice and Experience* 3, 6 (dec 1991), 655–666. DOI: http://dx.doi.org/10.1002/cpe.4330030610

[12] Simone Economo, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2016. Configurable and Efficient Memory Access Tracing via Selective Expression-Based x86 Binary Instrumentation. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 261–270. DOI: http://dx.doi.org/10.1109/MASCOTS.2016.69

[13] Josef Fleischmann and Philip A. Wilsey. 1995. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 50–58.

[14] Matteo Frigo and Steven G. Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, Vol. 3. 1381–1384. DOI: http://dx.doi.org/10.1109/ICASSP.1998.681704

[15] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the Multiconference on Distributed Simulation*. Society for Computer Simulation, 23–28.

[16] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, Michael Booth, and Fabrice Rossi. 2009. GNU Scientific Library Reference Manual. *Distribution* 954161734, March (2009), 592. DOI: http://dx.doi.org/ISBN0954612078

[17] Galen C. Hunt. 1998. Reversible load-time dynamic linking. (1998).

[18] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and System* 7, 3 (1985), 404–425.

[19] Justin M LaPre, Elsa J Gonsiorowski, and Christopher D Carothers. 2014. LORAIN: a step closer to the PDES 'holy grail'. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of Advanced Discrete Simulation (PADS)*. ACM Press, New York, New York, USA, 3–14. DOI: http://dx.doi.org/10.1145/2601381.2601397

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323. DOI: http://dx.doi.org/10.1145/355841.355848

[21] Yi-Bing Lin and Edward D Lazowska. 1990. *Reducing the saving overhead for Time Warp parallel simulation*. University of Washington Department of Computer Science and Engineering.

[22] Avinash C Palaniswamy and Philip A. Wilsey. 1993. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS)*. ACM, 127–134. DOI: http://dx.doi.org/10.1145/158459.158475

[23] Alessandro Pellegrini and Francesco Quaglia. 2014. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the Euro-Par 2013: Parallel Processing Workshops*, Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Constan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer (Eds.). LNCS, Springer-Verlag, 501–512. DOI: http://dx.doi.org/10.1007/978-3-642-54420-0_49

[24] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2009. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*. IEEE, 45–53. DOI: http://dx.doi.org/10.1109/PADS.2009.24

[25] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2015. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (2015), 1560–1569.

[26] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 135–148. DOI: http://dx.doi.org/10.1109/MICRO.2006.29

[27] Francesco Quaglia. 2001. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems* 12, 4 (2001), 346–362.

[28] Daniel Quinlan, Chunhua Liao, Justin Too, Robb Matzke, and Markus Schordan. 2013. ROSE Compiler Infrastructure. (2013). http://www.rosecompiler.org

[29] Robert Rönngren and Rassul Ayani. 1994. Adaptive Checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*. Society for Computer Simulation, 110–117.

[30] Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup, and Daniel Quinlan. 2015. Reverse Code Generation for Parallel Discrete Event Simulation. 95–110. DOI: http://dx.doi.org/10.1007/978-3-319-20860-2_6

[31] Markus Schordan, Tomas Oppelstrup, David R. Jefferson, Peter D. Barnes, and Daniel Quinlan. 2016. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application. In *Proceedings of the 2016 ACM-SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*. ACM Press.

[32] J. M. Shearer and M. A. Wolfe. 1985. ALGLIB, a simple symbol-manipulation package. *Commun. ACM* 28, 8 (aug 1985), 820–825. DOI: http://dx.doi.org/10.1145/4021.4023

[33] S Skold and Robert Rönngren. 1996. Event Sensitive State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference*. Society for Computer Simulation, 653–660.

[34] Marc Philippe Vertes. 2002. Method and system for managing shared-library executables. (2002).

[35] Robert Wahbe, Steven Lucco, and Susan L Graham. 1993. Practical Data Breakpoints: Design and Implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1–12.

[36] Darrin West and Kiran Panesar. 1996. Automatic Incremental State Saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS)*. IEEE Computer Society, 78–85.

[37] M. V. Zelkowitz. 1973. Reversible execution. *Commun. ACM* 16, 9 (1973), 566. DOI: http://dx.doi.org/10.1145/362342.362360

[38] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng Fai Wong. 2008. How to do a million watchpoints: Efficient Debugging using dynamic instrumentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4959 LNCS (2008), 147–162. DOI: http://dx.doi.org/10.1007/978-3-540-78791-4_10