Dealing with Reversibility of Shared Libraries in PDES

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

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

Markus Schordan
Lawrence Livermore National Laboratory
schordan1@llnl.gov

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

David R. Jefferson
Lawrence Livermore National Laboratory
jefferson6@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 a preliminary experimental evaluation, based on the integration of our library with the ROOT-Sim simulation engine.

1. INTRODUCTION

In Parallel Discrete Event Simulation (PDES) [15], various synchronization protocols have been proposed in the literature. Among them, the Time Warp speculative one [19] has been proven to be particularly effective, as it is relatively independent (in terms of its run-time dynamics) of both the simulation model's lookahead and the communication latency for exchanging data across simulation platform's threads/processes. 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 [5].

The speculative nature of Time Warp allows simulation events to be processed at any Logical Process (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* [19] 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*) [22, 6, 24, 31, 13, 35, 29] or by reducing the amount of data copied into a state snapshot (the so-called *incremental state saving*) [36, 26].

The reverse computing-based rollback operation, which tries to cancel the non-negligible memory footprint of the state saving technique, relies on *reverse events*, which can be generated either manually [7] or automatically [20, 8, 32, 33]. With respect to automatic generation of reverse events, the various proposals address it by relying either on binary instrumentation [20, 8] or on source-to-source transformation [32, 33]. 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 [34], GSL [16], FFTW [14], LAPACK [11], or BLAS [21] 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. This is something that speculative synchronization protocols, such as Time Warp, intentionally do not provide anytime. 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 particular, non-speculative applications have no need to rely on shared libraries enabled for reversibility, which can introduce a non-necessary overhead. Moreover, it would be required to instrument the whole

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGSIM-PADS '17, May 24–26, 2017, Singapore.
© 2017 ACM. ISBN 978-1-4503-4489-0/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3064911.3064927

shared library, even if only a small subportion is actually used by the simulation model, reducing the maintainability of the program.

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 [27]) and to a different behavior of the engine when dealing with model vs. platform code. Overall, by relying on the approach proposed in this paper, we enable speculative execution of any third-party library within a Time Warp-based simulation engine, significantly increasing the degree of programmability offered to simulation models' developers. This is done transparently for any combination of third-party libraries: libraries which rely on additional libraries are managed as well, thanks to the simple organization of our library-call detection system.

Our technique allows to be easily 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. No change to the simulation model's code is required for the integration. To assess the viability of our proposal, and to illustrate as well its operating simplicity, we present a preliminary experimental evaluation by integrating our approach with the ROme OpTimistic Simulator (ROOT-Sim) [25]. The experimental evaluation strives to assess the overhead introduced by our proposal when relying on third-party libraries in simulation models.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. Section 3 presents the design choices behind our proposal, and its implementation. The experimental assessment is 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 [20], 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 on any number of third-party libraries.

In the recent work in [32], the authors perform source-to-source transformation of C++ code based on the ROSE compiler infrastructure [30], intercepting all operations which modify memory 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 the 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 [36, 26], 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 our work, the authors in [8] do not account for the presence of third-party libraries, therefore limiting the degree of programmability of simulation models. In particular, the technique presented in [8] could not be used out-of-the-box to instrument third-party libraries. Indeed, static binary instrumentation cannot be used at runtime to intercept generic library calls. This is exactly the focus of this paper.

An approach similar in spirit to our proposal is the one of Valgrind [23]. This framework gains control of the program's execution in a way similar to what we do, in order to generate (in a JIT fashion) instrumented versions of the program code. Nevertheless, since the goal of Valgrind is debugging, it relies on a virtualized CPU which runs the client program. This introduces a slowdown which can be afforded when debugging, but cannot be tolerated when running high-performance simulations. Conversely, we generate instrumented code which is run by physical cores, and interacts with the library only when needed, without relying on any kind of virtualization.

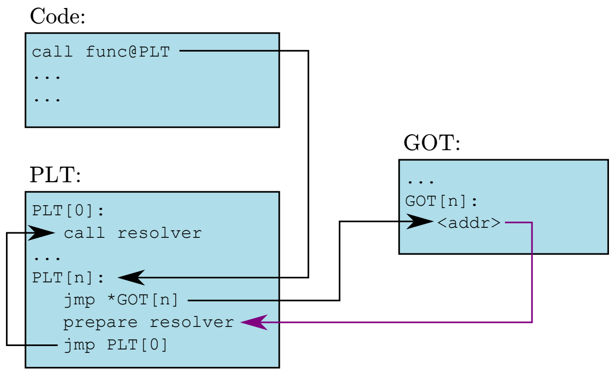
Our proposal is also related to a number of works in the field of program execution tracing (see, e.g., [1, 4, 28, 38]) for debugging, vulnerability assessment and repeatability. These approaches provide detailed analysis of changes in the state of the program, and of the execution flow. Nevertheless, 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 [18] explicitly deals with reversibility of shared libraries within executables. Yet, differently from our proposal, the goal 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.

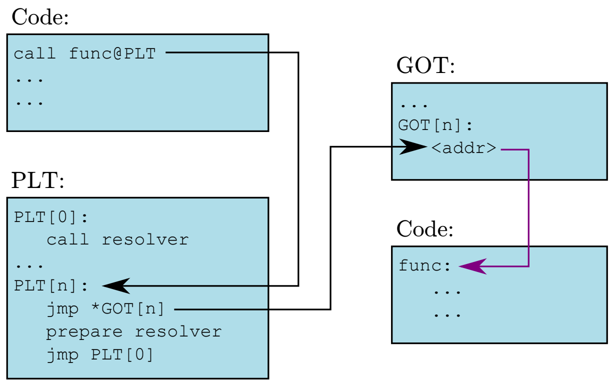
3. SHARED LIBRARIES REVERSIBILITY

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 reference 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 function towards its actual implementation. In particular, the compiler:

- Records 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



(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.

is moved to a different environment where a compatible library's version is not present, the loader fails to resolve any call, so as to avoid undefined behaviors;

- Reserves an entry in a special table, called the *Procedure Linkage Table* (PLT), 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;
- 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 dynamic loading. 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. If it is so, the library symbol is resolved, otherwise the (already-resolved) function is simply called.

This is exactly where our proposal acts so as to generate reversibility-enabled copies of third-party library functions. To illustrate this mechanism, 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(a). 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 it 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 call `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 alters 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 module 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;
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]`;

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

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 of Time Warp-based simulations, necessarily to support the rollback operation. All these points above demand special care, and we therefore describe in the following how they are supported by `libreverse`.

3.1 Intercepting 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 respect 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 use the following strategy 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, 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()` call, making it both writable and executable. We then copy the whole content of the functions' binary representation in it. This will be its working copy, 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. The second one relates to identifying, among all the instructions, those which write on memory. These instructions should be properly altered to generate *reverse instructions* on the fly, 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 to its actual opcode. This opcode is matched against 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

[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

(which in case of a simple `mov` instruction 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 to prepare 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 [2, 3] 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 *caller-save* registers, after having pushed 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. In this way, 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 $base\ address + (index * scale) + displacement$. The parameters *scale* and *displacement* are already encoded in the instruction binary representation, while *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.

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 referenced 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. Indeed, 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 from 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.

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 particu-

[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)

lar, this disassembler sets a global (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) 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 might be shorter 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 in 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*. This index keeps, for each instruction, its size in bytes. Therefore, if the end of the function is reached while coalescing instructions, it is possible to

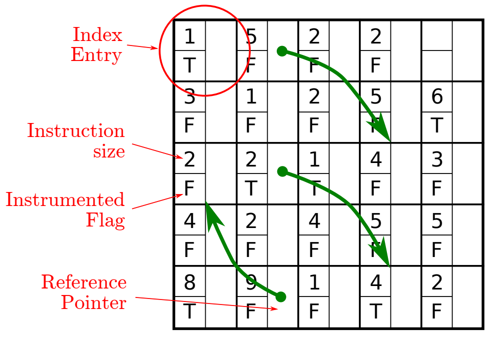


Figure 4: The index of assembly instructions built while instrumenting a library function.

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 code snippet:

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

The instrumentation process will detect that the `movs` is a memory-write instruction, and 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 gives 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, adding (for each instruction) a reference. 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 instrumentation 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 final organization of the instruction index is depicted in Figure 4.

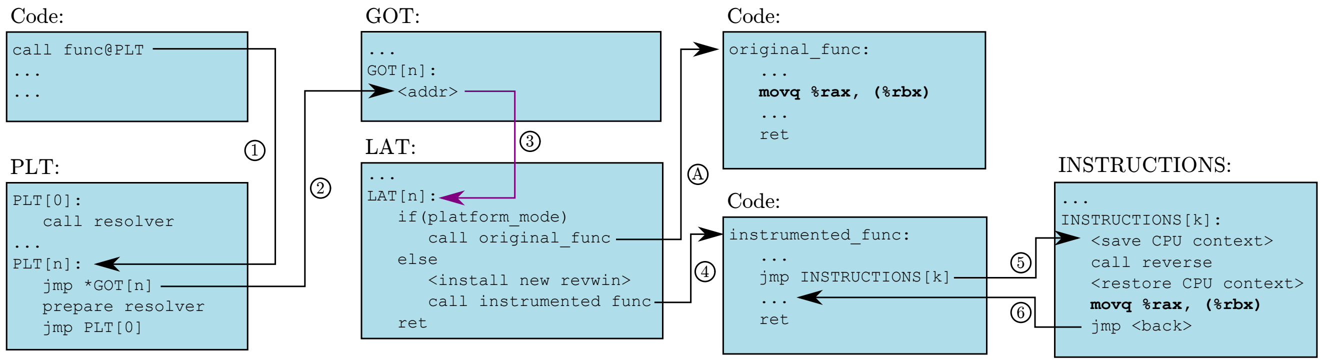
The instructions index becomes handy to solve a couple of additional issues, related to the way shared libraries are built. In particular, any library function within a library can reference any other function within the library itself. Since a `call` instruction, to invoke another function in the library, 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 `call`s 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 a library as complex as `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. Future work entails generalizing the approach, in order to specifically deal with this corner case with any third-party library.

To conclude, at the end of the instrumentation process, the organization of the memory map to execute a call to a library function is depicted in Figure 5.



```
call func@PLT
...
...
```

```
PLT[0]:
   call resolver
...
PLT[n]:
   jmp *GOT[n]
   prepare resolver
   jmp PLT[0]
```

```
...
GOT[n]:
   <addr>
```

```
...
LAT[n]:
   if(platform_mode)
      call original_func
   else
      <install new revwin>
      call instrumented func
   ret
```

```
original_func:
   ...
   movq %rax, (%rbx)
   ...
   ret
```

```
instrumented_func:
   ...
   jmp INSTRUCTIONS[k]
   ...
   ret
```

```
...
INSTRUCTIONS[k]:
   <save CPU context>
   call reverse
   <restore CPU context>
   movq %rax, (%rbx)
   jmp <back>
```

Figure 5: Organization of code, tables, and trampolines after instrumentation. ①: the application invokes the library function `func`, by calling into `PLT[n]`; ②: the program jumps to the corresponding LAT entry by dereferencing the n-th pointer in the GOT (③); Ⓐ: if in platform mode, the trampoline activates the original version of `func`, otherwise ④ the instrumented one; ⑤: instructions moved to the `INSTRUCTIONS` table are accessed via jumps; ⑥: the instrumented function regains control.

3.3 Generation and Management of Reverse Instructions

The instrumentation mechanism 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.

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., [26]), 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 addition, differently from incremental state saving, our organization of reverse instructions has a direct positive impact on the restore operation. In fact, traditional incremental state saving approaches (see, e.g., [27]) require scanning the metadata table to determine which portions of the simulation state should be restored from a certain snapshot, and possibly require to generate temporary additional metadata during this operation. By relying on dynamically-generated reverse instructions, we do not incur into this cost.

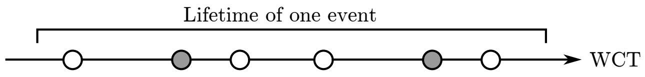


Figure 6: Interleaved memory updates within one event. White circles refer to memory updates directly made by the model's code. Grey circles refer to memory updates made through calls to shared library functions.

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* and *library call incarnation*. `libreverse` organizes reverse instructions in atomic blocks, on a per-thread fashion, in a way similar to the work presented in [8]. Every time that a new reverse instruction is generated, it is inserted in a stack of instructions, 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`). Since the focus of this paper is on how to efficiently and effectively enable generic third-party libraries for reversibility, we will not describe in details reverse instructions are generated and managed. For a thorough description of this aspect, we refer the reader to the previous work in [8].

Nevertheless, packing together all the reverse instructions generated within the same event could lead to erroneous reconstructions of a previous state upon the execution of a rollback operation. In fact, if we look at the example in Figure 6, the same memory buffer could be updated in an interleaved fashion by calls to library functions and direct memory updates by the simulation model. Executing all re-

[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.

verse instructions generated by library functions as a whole, in fact, might not respect the ordering of updates executed in forward execution. To this end, `libreverse` uses a reverse window for each library call invocation, which is detected since the execution flow has to pass always through the corresponding LAT entry.

To let the simulation engine keep control of generated reverse windows on a per-event basis, `libreverse` exposes an API function named `finalize_event()` which, upon invocation, returns a list of all reverse windows generated since the last call to `finalize_event()` itself. The simulation engine can then link this set to any representation of the event in the just-descheduled LP's 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 set of reverse windows in case, e.g., an event is deemed committed or it is removed from the message queue due to the reception of an antimessage. To determine the proper execution order of reverse windows, each of them is stamped with a unique mark, based on the value of the Time Stamp Counter (TSC) x86 register. `libreverse` exposes the `get_mark()` API function (based on the `rdtsc` assembly instruction) to deliver values from TSC to the simulation engine. In this way, the engine can stamp as well portion of its data/metadata used to enforce reversibility of the model, determining in this way the proper execution order of reverse operations globally, for each event to be undone. By relying on this set of API functions, any Time Warp-based simulation engine can be easily integrated with `libreverse`.

As a last note, we should discuss how we deal with updates of *local variables* of library functions, which live in the stack of the instrumented function. In case a functions call is undone due to a rollback operation, we do not need to revert changes to these variables—they are ephemeral with respect to the scope of the function—and therefore we do not need to generate reverse instructions for these. To this end, we adopt the following approach to filter out stack updates, trying to reduce as well the overhead introduced at runtime. In particular, when disassembling an instruction, we check whether the addressing mode involves any displacement from the stack pointer or the base pointer, which are related to access to local variables. At the same time, we can easily check whether a memory-update operation involves the management of the stack (e.g., in case of `push` or `pop` instructions). In both cases, `libreverse` does not instrument these memory-access instructions, as they are related to data which is not necessary to reverse. Nevertheless, nothing prevents a program to pass as a reference a variable on stack. In this case, it is not possible to check the on-stack condition statically. Therefore, whenever the `reverse` function gains control, it checks whether the computed destination address of the memory write falls within the stack—this can be easily done by comparing with the current value of the stack pointer register. If it is so, `libreverse` simply returns, without generating any reverse instruction. We note that in this latter case we pay an additional overhead to compute the target address, but we keep consistency of the approach even under complex stack-usage patterns.

3.4 Dealing with Memory Allocations and Deallocations

The last aspect to be dealt with in order 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 [27], as providing a memory manager is out of the scope of our approach. Therefore, in order to correctly connect `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)` 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

4.1 Test-bed Environment

We have integrated `libreverse` within the ROOT-Sim simulation platform[5] [25]. This is a C-based open source simulation package targeted at POSIX systems, which implements a general-purpose simulation environment based on the Time Warp synchronization paradigm. It offers a very simple programming model relying on the classical notion of simulation-event handlers (both for processing events and for accessing a committed and globally-consistent state image upon GVT calculations), to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution. It supports the execution of the rollback operation either via traditional checkpointing facilities [27], or via reverse code blocks [8].

Our experiments have been run on top of a 32-core HP ProLiant server equipped with 64GB of RAM and running Debian 6 on top of the 2.6.32-5-amd64 Linux kernel. This is a common setup for HPC applications, as this is a software configuration which offers a very good tradeoff between the services exposed to user space applications and performance. At the same time, our approach is so general that it can be used as well on more modern environments.

In order to integrate `libreverse` with ROOT-Sim, we have slightly touched two different aspects of the simulation engine: the models' compilation toolchain (which relies on the `rootsim-cc` custom compiler) and the reversibility facilities

[5]ROOT-Sim is available at http://github.com/HPDCS/ROOT-Sim.

already present in the engine [8]. As for the compilation toolchain, we have only statically linked the final executable against `libreverse`, allowing the program constructor described in Section 3 to take control before the actual simulation engine is started. This allows to setup the patched dynamic linker's resolver which will be activated whenever an invocation to a library function is issued.

On the other hand, ROOT-Sim's reversibility facilities presented in [8] rely on an instrumentation mechanism similar in spirit to the one used by `libreverse`. In particular, at compile time, `ROOT-Sim` performs a static binary instrumentation process, picking all the memory-update operations that are located in the simulation model's code. These instructions are intercepted by the memory map manager, which generate undo code blocks to reverse the effects of the execution of the event on memory. `libreverse`, on the other hand, keeps its reverse instructions in a separate buffer, as described in Section 3.3.

To ensure that the reverse execution is performed in the proper order, we rely on the `get_mark()` API exposed by `libreverse`, so that reverse code generated by ROOT-Sim can be tagged with data which allow to determine the total order of reverse actions to take. In this way, upon a rollback execution, ROOT-Sim is able to undo the effects on memory by the forward execution of events by either relying on its internal reversibility management, or invoking the `execute_revwin()` function.

4.2 Test-Bed Application Model

As a test-bed application, we rely on the Sensors Network Model (SNM), which simulate the behavior of wireless sensor networks (WSN). WSN are networks composed of small devices featuring power source, a microprocessor, a wireless interface, some memory and one or more sensors. They are used to gather information in a given location or region, yet due to the limited radio communication range, nodes can communicate using multi-hop routing protocols.

SNM implements the Collection Tree Protocol (CTP) [17] to collect data from wireless sensors networks. In particular, it relies on a variant of the library offered by TinyOS [9]. CTP is a distance vector routing protocol, which computes the routes from each node in the network to the root (specified destinations) in the network. Each node forwards packets to its parent, chosen among its neighbor nodes. In order to make a choice, each node must be aware of the state of its neighbors: that's why nodes continuously broadcast special packets, called *beacons*, describing their condition.

The metric adopted in CTP for the selection of the parent node is the *Expected Transmissions* (ETX). A node whose ETX is equal to n can deliver a data packet to the root node with an average of n transmissions. The ETX of any node is recursively defined as the ETX of its parent plus the ETX of its link to the parent; the root node represents the base case in this recursion, and its ETX is obviously equal to zero.

CTP uses these three mechanisms to overcome the challenges faced by distance vector routing protocol in a highly dynamic wireless network: i) the *link estimator*, which is in charge of computing incoming and outgoing quality of the links; ii) the *routing engine*, which is dedicated to the selection of the parent node, i.e. the neighbor with the lowest value of the multi-hop ETX, and iii) the *forwarding engine*, which forwards data packets, detects and tries to fix routing loops, and detects and drops duplicate packets.

It is interesting to note that the routing engine has to maintain a table, called routing table, where it stores the last ETX value read in the beacons from each neighbor. In this way, it is able to always choose the "best" neighbor (the one with the lowest multi-hop ETX) as parent. Therefore, it has to continuously update the table reading the information contained in the beacons received from the neighbors. In the simulation model, each LP represents a wireless sensor. The routing table managed by the routing engine (along with other data structures related to all three components of the CTP protocol) is kept within the LP's simulation state. Upon the reception of a simulation event representing a beacon, the simulation model passes the received information to the CTP library, which recomputes all parameters related to the network and then updates the routing table in the LP state, thus performing a set of memory updates. These memory updates are intercepted by `libreverse`, which therefore enables for reversibility the CTP library.

4.3 Experimental Data

In order to assess the overhead introduced by our proposal, we have relied on experiments using two variants of the model. In one variant, the CTP algorithm is dynamically linked to the simulation model, while in the other, the library's code is directly incorporated into the simulation model. In this latter configuration, the actions to correctly restore a previous checkpoint are completely demanded from ROOT-Sim. As mentioned, ROOT-Sim supports the rollback operation via both checkpointing facilities and reversibility facilities. Therefore, in the experiments we present data related to reversibility obtained when having the CTP library linked against the executable (LIB in the plots), by relying on the reversibility facilities offered by ROOT-Sim (REV in the plots), and by relying on traditional sparse state saving (CKPT in the plots), with the checkpointing interval optimized according to the results in [27].

We have run two sets of experiments. In one experiment, we have set the total number of sensors to 300, while in a second one, we have set it to 2000. At the same time, the area in which the sensors are deployed is kept fixed in both configuration, thus having a denser concentration of sensors in the second setup. In both configurations, the sensors are randomly placed within a square region. Since the CTP algorithm keeps a routing table within each LP's state, and since it is updated whenever a beacon message is received, the denser scenario has a twofold effect: i) the size of the state of each LP is larger, and ii) the frequency of state update by the CTP library is increased. Indeed, since when the number of sensors is increased to 2000, the average distance between two sensors decreases, so the number of beacons that can travel the transmission channel without being affected by fading effects is higher. All the experimental results are averaged over 10 different runs.

In Figure 7, we report experimental data when the model is run using 300 sensors. By the results, we can see that the configuration presenting the best performance profile is the one associated with reversible execution managed natively by ROOT-Sim. This is mainly due to the fact that the memory-update profile associated with this configuration does not entail a large number of memory updates. When using `libreverse`, there is an overhead around 7%,

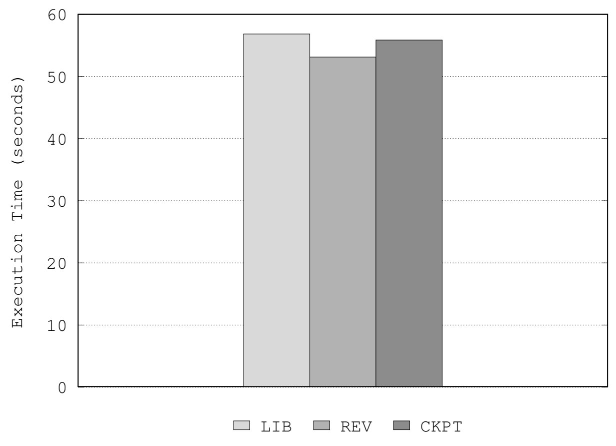


Figure 7: Results with 300 sensors.

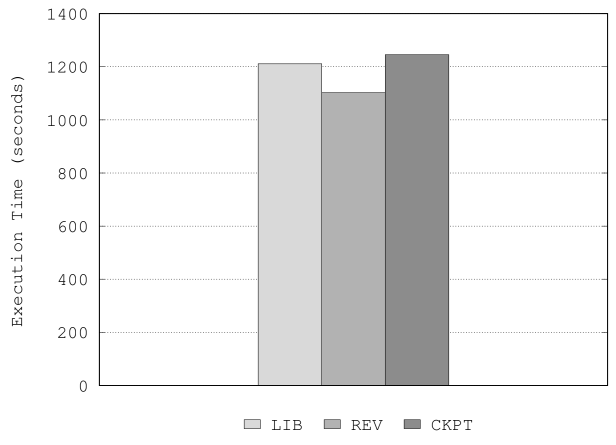


Figure 8: Results with 2000 sensors.

while when relying on checkpointing there is an overhead around 4%, which are both anyhow negligible.

The overhead introduced by `libreverse` is mostly related to the fact that the simulation model performs memory updates as well. Therefore, since the amount of data written by the CTP shared library is not very large, the time spent by the simulation engine to switch among the two reversibility approaches in order to guarantee the correct order of the actions is not paid off. At the same time, the overhead of `libreverse` is slightly higher than that of the checkpoint-based state restore exactly because both the size of the state and the amount of data updated in it is reduced, thus optimized checkpointing facilities (especially is realized according to autonomic facilities as in [27]) are able to fine tune themselves significantly. In addition to this, the rollback length is reduced—sensors are sparse, the communication is organized according to a tree, and therefore the probability of cascading rollbacks are small, and the number of LPs that can rollback each other is limited. In this scenario, plain reverse computation is not surprisingly delivering a better performance.

Figure 8 reports experimental data when running the configuration with 2000 sensors. In this configuration, the results slightly change: reverse computation managed by ROOT-Sim still has the better performance result, checkpoint-based executions have the highest performance penalty (more than 12%), while `libreverse`-based reversibility has a overhead slightly smaller than 10%. The trend inversion among checkpoint based and reversibility based is due to the fact that, since the amount of state updates is non minimal, the checkpointing system has to restore a large amount of data, either in a full or incremental fashion. At the same time, the performance gain by the reversibility engine internal to ROOT-Sim is again related to the fact that in this case there is no need for the continuous switch to guarantee the correct order of the reversibility actions.

Overall, the overhead introduced by relying on `libreverse` is not extremely high (10% in the worst case), considering that much overhead is introduced by the fact that the simulation engine has the burden of ensuring the proper order of execution of reverse activities. What is most important, anyhow, is that the approach proposed allows to enable reversibility of generic third-party shared libraries in a completely transparent manner towards the application level code, closing the circle of automatic reversibility in the context of speculative Time Warp-based simulation.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have discussed the design and implementation of `libreverse`, which allows to create on the fly instrumented versions of functions offered by any third-party library. By relying on this approach, it is possible to enable for reversibility any shared library, making them compliant with the Time Warp optimistic synchronization protocol. The experimental assessment has shown that it is possible to increase the programmability degree of PDES models, exactly relying on services offered by shared libraries, paying only a little overhead.

Future work entails the design of a reversible memory map manager, which can offer to the simulation engine (or, in general, to any program) a memory map on which effects of atomic actions (such as the execution of an event) can be reverted. This approach will drastically reduce the overhead due to repeatedly switching between reversibility actions. Additionally, it will be possible to rely on incremental checkpointing and reverse computing based both on binary instrumentation and source to source transformation, to any degree of integration. Similarly, a thorough experimental evaluation of the trade-offs between these approaches will be topic of future work.

Acknowledgements

This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, Lawrence Livermore National Security, LLC. IM release number LLNL-CONF-723277.

The authors warmly thank Davide Leoni for his implementation of SNM which has been used in this work for the experimental assessment.

6. REFERENCES

[1] GDB: The GNU Project Debugger.
[2] System V Application Binary Interface, Intel386 Architecture Processor Supplement, 1997.

[3] System V Application Binary Interface AMD64 Architecture Processor Supplement, 2007.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Notices*, 35(5):1–12, 2000.

[5] P. D. Barnes, C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of advanced discrete simulation*, PADS, pages 327–336, 2013. ACM Press.

[6] S. Bellenot. State skipping performance with the Time Warp operating system. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, PADS, pages 53–64, 1992. ACM Press.

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

[8] D. Cingolani, A. Pellegrini, and F. Quaglia. 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, 2015. ACM Press.

[9] U. Colesanti and S. Santini. The collection tree protocol for the castalia wireless sensor networks simulator. Technical report, ETH Zurich, 2011.

[10] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, pages 142–156. Springer-Verlag, 2006.

[11] J. Demmel. LAPACK: A portable linear algebra library for high-performance computers. *Concurrency: Practice and Experience*, 3(6):655–666, dec 1991.

[12] S. Economo, D. Cingolani, A. Pellegrini, and F. Quaglia. Configurable and Efficient Memory Access Tracing via Selective Expression-Based x86 Binary Instrumentation. In *Proceedings of the 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 261–270. IEEE Computer Society, 2016.

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

[14] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceesings of the International Conference on Acoustics, Speech and Signal Processing*, ICASSP, pages 1381–1384, 1998.

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

[16] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. GNU Scientific Library Reference Manual. *Distribution*, 954161734:592, 2009.

[17] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of the 7th Conference on Embedded Networked Sensor Systems*, 2009.

[18] G. C. Hunt. Reversible load-time dynamic linking, 1998.

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

[20] J. M. LaPre, E. J. Gonsiorowski, and C. D. Carothers. LORAIN: a step closer to the PDES 'holy grail'. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pages 3–14, New York, USA, 2014. ACM Press.

[21] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[22] Y.-B. Lin and E. D. Lazowska. *Reducing the saving overhead for Time Warp parallel simulation.* University of Washington Department of Computer Science and Engineering, 1990.

[23] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 47–69, 2003.

[24] A. C. Palaniswamy and P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, PADS, pages 127–134. ACM Press, 1993.

[25] A. Pellegrini and F. Quaglia. The ROme OpTimistic Simulator: A tutorial. In D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Constan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, editors, *Proceedings of the Euro-Par 2013: Parallel Processing Workshops*, PADABS, pages 501–512. LNCS, Springer-Verlag, 2014.

[26] A. Pellegrini, R. Vitali, and F. Quaglia. 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*, pages 45–53. IEEE, 2009.

[27] A. Pellegrini, R. Vitali, and F. Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1560–1569, 2015.

[28] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

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

[30] D. Quinlan, C. Liao, J. Too, R. Matzke, and M. Schordan. ROSE Compiler Infrastructure, 2013.

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

[32] M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan. Reverse Code Generation for Parallel Discrete Event Simulation. pages 95–110. 2015.

[33] M. Schordan, T. Oppelstrup, D. R. Jefferson, P. D. Barnes, and D. Quinlan. 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, 2016.

[34] J. M. Shearer and M. A. Wolfe. ALGLIB, a simple symbol-manipulation package. *Communications of the ACM*, 28(8):820–825, aug 1985.

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

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

[37] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, 1973.

[38] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W. F. Wong. 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:147–162, 2008.