# Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES

Davide Cingolani, Alessandro Pellegrini, Francesco Quaglia
DIAG – Sapienza, University of Rome
Via Ariosto 25, 00185 Rome, Italy
cingodvd@gmail.com, {pellegrini, quaglia}@dis.uniroma1.it

## ABSTRACT

The rollback operation is a fundamental building block to support the correct execution of a speculative Time Warp-based Parallel Discrete Event Simulation. In the literature, several solutions to reduce the execution cost of this operation have been proposed, either based on the creation of a checkpoint of previous simulation state images, or on the execution of negative copies of simulation events which are able to undo the updates on the state. In this paper, we explore the practical design and implementation of a state recoverability technique which allows to restore a previous simulation state either relying on checkpointing or on the reverse execution of the state updates occurred while processing events in forward mode. Differently from other proposals, we address the issue of executing backward updates in a fully-transparent and event granularity-independent way, by relying on static software instrumentation (targeting the x86 architecture and Linux systems) to generate at runtime *reverse update code blocks* (not to be confused with *reverse events*, proper of the reverse computing approach). These are able to undo the effects of a forward execution while minimizing the cost of the undo operation. We also present experimental results related to our implementation, which is released as free software and fully integrated into the open source ROOT-Sim (ROme OpTimistic Simulator) package. The experimental data support the viability and effectiveness of our proposal.

## Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event, Parallel*; D.3.4 [**Programming Languages**]: Processors—*Code Generation*

## General Terms

Algorithms, Performance

*SIGSIM-PADS'15,* June 10–12, 2015, London, United Kingdom.
Copyright © 2015 ACM ISBN 978-1-4503-3583-6/15/06...$15.00.
DOI: http://dx.doi.org/10.1145/2769458.2769482.

## Keywords

PDES; Speculative Processing; Code Instrumentation; Reversibility

## 1. INTRODUCTION

In Parallel Discrete Event Simulation (PDES) [11], Time Warp [12] has been proven to be an effective synchronization protocol, which has been shown to be 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. All these peculiarities allow it to guarantee high performance even in systems that are not tightly coupled and/or possibly entail up to millions of processors [3]. According to classical PDES, in Time Warp the simulation model is partitioned into distinct simulation objects, which are mapped to Logical Processes (LPs). The latter are in charge of handling the execution of impulsive events, which ultimately produce state updates (hence transitions) in the actual simulation model state.

Time Warp is a speculative protocol, hence it allows simulation events to be processed at any LP optimistically. This means that they are processed 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.

In the literature, this operation has been thoroughly studied. Different approaches have been proposed, which provide benefits in differentiated scenarios. All these solutions can be mainly grouped into two separate families, namely *checkpoint-based* [12] and *reverse computing-based* [6], depending on the algorithmic technique which is used to bring one simulation state to a previous (consistent) snapshot.

The checkpoint-based rollback operation grounds on the possibility, for a simulation engine, to know the location (in main memory) of each simulation object's state. By exploiting this information, the engine can therefore create a copy of the simulation state after the execution of one (or a group of) operations which have produced state updates. In this context, different possibilities have been presented, all aiming at reducing the cost (both in terms of memory and CPU usage) paid to create a state snapshot, which will be (possibly) used for a later state restore. Among the various research lines, we find two main different approaches, which have been often combined together. On the one hand, we find solutions to reduce the frequency according to which a simulation object's state is inspected to create a copy—the so-called *sparse* or *periodic state saving* [15, 4, 16, 26, 9, 28, 24]—with a focus on detecting which is the best-suited checkpointing interval to minimize unfruitful work (e.g., taking checkpoints which are never used for a rollback operation). On the other hand, we find solutions which try to reduce the amount of data copied into a state snapshot, ensuring anyhow that no meaningful piece of information is lost at any time—the so-called *incremental state saving* [35, 19]. A mixture of these approaches has been proposed as well in [21], trying to modify at runtime the execution mode of the state saving operation, depending on the current execution dynamics, in order to capture different execution phases of the simulation models.

At different scales, all these solutions suffer from the high cost associated with making a (logically) complete copy of the simulation state, which is either proportional to the size of the state (in case of full state saving) or to the number of update operations related to the execution of one or more events (in case of incremental state saving).

The reverse computing-based rollback operation, on the other hand, tries to cancel the non-negligible memory footprint produced by the state saving technique. This solution grounds on the availability (either on a manual [6] or automatic [14] basis) of reverse copies of simulation events, such that if the execution of a forward event $e$ on a simulation state $S$ produces a state transition $e(S) \rightarrow S'$, then the execution of the reverse event $r$ associated with $e$ on $S'$ produces the inverse transition $r(S') \rightarrow S$. Overall, while the reverse computing approach is able to strongly reduce the impact of memory usage from which state saving may suffer, the execution cost of the rollback operation is directly proportional to the execution time of simulation events. This cost could become predominant in case of events with a high granularity, and in case the rollback length is non-minimal.

By mixing the different philosophies standing behind the above state recoverability techniques, we present a new approach which is based on the combination of undo-logs with software reversibility. Particularly, in our proposal, the data that are typically recorded by undo-log systems are used to generate so called *update undo code blocks*. The latter can be used to squash the memory side effects (namely updates) generated by a non casually-consistent portion of the computation.

To achieve our goal, we rely on ad-hoc software instrumentation, which allows us to capture the *effects* on memory of an event's forward execution. This information is then used to build at runtime the update undo code blocks, which are basically compact undo logs of actual operations. They are compact because they are specifically encoded directly as a set of subsequent machine instructions which are able to cancel the overall effects of the execution of forward events. This is different from the reverse computing technique, as our approach is independent of the actual event granularity. In fact, the execution cost of an update undo code block is only proportional to the amount of memory areas touched in write mode during the forward execution.

Further, our approach is different from classical undo logs used in, e.g., the context of fault tolerance. In fact, undo log architectures are not based on dynamic generation of reversibility code. Also, contrarily to incremental state saving, our solution is specifically designed to avoid the (high) cost paid by incremental state saving when executing a rollback operation. In fact, while the latter generally requires to inspect some (arbitrarily complex) metadata to determine where each portion of the incremental log should be placed in memory, our update undo code blocks can be simply executed with no additional overhead associated with metadata management, as they are specifically generated so as to keep all the relevant information already packed together.

Another important aspect in our proposal is that it can be combined with classical checkpointing methods (both incremental and non-incremental). Particularly, by taking infrequent checkpoints, we can generate checkpoint intervals where some passed-through snapshots are not coverable by update undo code blocks, while others are. Hence state restoration can be executed by either reloading a previous checkpoint and executing a classical coasting forward phase, or by applying the update undo code from the current state snapshot (or from a conveniently selected checkpointed snapshot). While the former case occurs for restoration to a snapshot uncovered by the update undo facility, the latter occurs for snapshots covered by the update undo scheme. In other words, a single checkpoint interval can be optimized by the combination of the two techniques, in terms of the tradeoff between overhead for recoverability tasks and actual recovery costs. Other proposals do not have such a fine-grained mixture of different recoverability supports. In fact, most mixed-based solutions either switch between different recoverability modes along different phases of the simulation model execution (i.e., not in a single checkpoint interval) [21], or are exclusively based on incremental vs non-incremental checkpointing, thus not relying on the update undo code block technique.

Finally, borrowing from the results in [7], we present an analytic model that allows to determine how to partition each single checkpoint interval in terms of recoverability support, thus optimizing the decision on the length of the interval, as well as the decision on what are the snapshots which will be reconstructed (in case of a rollback operation) by relying on the coasting forward operation or on update undo code blocks.

We again remark that our proposal is fully application transparent, and has been released as free software and integrated into the open source ROOT-Sim package [18, 20].

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

As already pointed out, due to the fundamental role of the rollback operation in the context of optimistic synchronization for PDES, the literature on this topic is extremely wide. In the state saving-based context, several solutions have been introduced for logging the whole state of a simulation object (at each event execution or after an interval of executed events) [9, 22, 25, 26], or incrementally logging modified state portions [27, 31, 35], or supporting a mix of the two approaches [10, 29]. These solutions either ask the application-model developer to implement callbacks which are explicitly invoked when the simulation environment determines that a state log is necessary, or require the modeler to issue a call to some specific API used to identify the simulation state location in memory, or request to statically identify (e.g. at compile-time) which portions of the address space need to be considered part of the state. The works in [8, 30] address the management of dynamic memory maps to store simulation objects' states, and in [32] the management of dynamic memory is additionally tackled in a completely transparent way, by proposing a memory manager specifically targeted at optimistic PDES environments, which allows to transparently manage a dynamically scattered memory map of simulation objects' states. Our proposal is complementary to all the aforementioned ones, as we integrate traditional state saving with the runtime generation of update undo code blocks.

In the context of reverse computing-based state restore, a recent work [14] presents a software-instrumentation based approach (at the level of LLVM IR) to automatically generate the code associated with negative events. This work is similar in spirit to ours, as one of our final goals is to relieve the user from the burden of implementing the negative version of the events as well. Nevertheless, in the approach in [14], binary instrumentation is used to generate at compile time exact negated versions of code blocks, while we generate at runtime the instructions which undo the effects of the execution of one event in memory. Therefore, our solution's cost (in terms of recovery latency) is not proportional to the granularity of the events, rather to the amount of memory locations which are updated during the execution in forward mode. On the other hand, the final tradeoff by our solution is different from the one in [14] also because our instrumentation scheme operates at run-time, thus inducing some cost for the dynamic generation of update undo code blocks.

The work in [19], similarly to what we do, relies on static binary instrumentation to track memory updates during the forward execution of the events. Nevertheless, the goal in [19] is to use this information to generate periodic incremental checkpoints. Contrarily, we use tracked memory updates to build update undo code blocks. This is similar, as well, to the proposal in [35], but rather than packing the undo log in linked data structures, we pack on the fly assembly instructions which are later executed consecutively, so as to reduce to the highest extent the execution time of the restore operation.

Our approach shares underlying principles with the works in [7, 21]. In [21], an autonomic system to determine at runtime the best suited checkpointing mode (incremental vs full) is presented. Efficiency is ensured by relying on a dual-version executable technique, which allows to switch between the two execution modes changing only a couple of function pointers. We keep the same ability to change the support for state recoverability (using the same software dual-version technique), but we do this within each single checkpoint interval (by covering a subset of the states passed through in the interval by update undo code blocks). Hence we are able to optimize dynamics at smaller granularity levels (single events within a checkpoint interval). With respect to [7], we propose a similar model. However, in that work no support for application transparency of recoverability tasks is presented, while our proposal is fully application transparent.

Our proposal is also related to a number of works in the field of program execution tracing (see, e.g., [1, 2, 23, 36]) for debugging, vulnerability assessment and repeatability. These approaches provide detailed analysis of changes in the state of the program, and of the execution flow. However, this is achieved via performance-intrusive techniques relying on dynamic instrumentation and/or kernel-level services, unsuited for contexts where performance cannot be sacrificed (e.g., parallel simulation). Debugging supports showing basic operating modes comparable to ours (namely, the employment of trap mechanisms based on code insertion and/or replacement to detect memory write accesses) are those addressing data watch points (see, e.g., [34]). However they have performance targets different from ours since optimizations mostly cope with search techniques for verifying whether a memory reference falls inside a region that is currently subject to a watch point. In other words, aspects related to the identification of areas that have been modified and to log/restore operations are not considered.

## 3. THE STATE RECOVERABILITY ARCHITECTURE

Our architecture to manage state recoverability relies on static software instrumentation to transparently modify the application-level code, in order to let the simulation engine track at runtime what are the *effects* of the forward execution of events on the simulation model's state. This information is used to build a packed version of negative instructions which only undo the effects of the forward execution, allowing for a reverse execution of events (in terms of state updates) which is independent of the actual forward event granularity. This technique is complemented with an analytic model which determines, during the forward phase, whether the next event updates will be guaranteed to be recoverable either through classical coasting forward, or via the execution of update undo code blocks. The model decisions are aimed at minimizing an overhead function that expresses the tradeoff between the cost of recoverability tasks and actual recovery operations. The whole approach will be based on the coexistence of dual executable modes (inspired to [21]). This allows to quickly switch from an instrumented version of the simulation model's code (which allows to track memory updates and to generate update undo code blocks) to the "plain" one (where no actual tracking is performed). In this section, we will describe the various design choices which have driven the implementation of our proposal.

### 3.1 Instrumentation Technique

To statically instrument the application-level code, we rely on Hijacker [17], an open-source static instrumentation tool specifically targeted at HPC applications. Hijacker has been developed on the first instance to support incremental state saving in optimistic PDES systems [19], but has later been extended to support differentiated tasks. For the proposal in this paper, we have augmented the set of operations which this tool can perform. The basic support provided by Hijacker is to let the user specify (by using simple xml-based rules) what are the operations to perform on the original application code. Hijacker is conceived to be part of the compiling tool-chain, placing itself as a pre-linking stage. This allows our state recoverability architecture to be easily integrated into different simulation engines.

Hijacker works on *relocatable object files*. Specifically, it operates on the Executable and Linkable Format (ELF). The rules interpreted by Hijacker allow to perform instrumentation at different scopes, namely executable-wide or at the

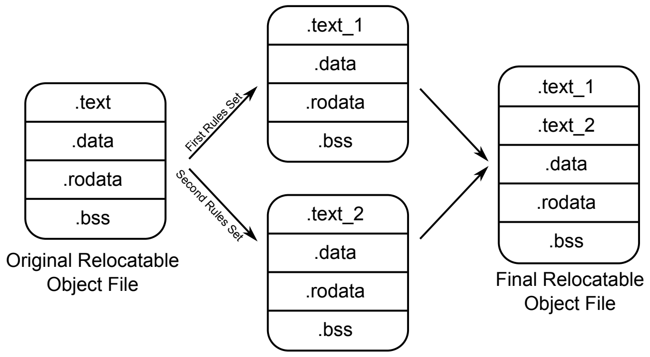

**Figure 1: Code multiversioning**

level of single functions/instructions. Additionally, the rules allow to instruct Hijacker to create multiple copies of the same executable, but differently instrumented. This technique, known as *multi-coding*, creates different versions of the code which nevertheless share the same data sections within the virtual address space. Hijacker transparently allows to change the name of all the (instrumented) functions when it comes to generate multiple versions of the software, by simply appending a user-defined suffix to them. This allows the simulation engine to exactly identify which copy of the application-level function is instrumented in a specific way. Overall, after the generation of multiple versions of the code, the final memory organization of the executable complies with the scheme shown in Figure 1.

To reach our target, we have specified the rules reported in Figure 2 and implemented the specific modules that apply them to the original executable. This configuration file instructs Hijacker to generate a modified application-level relocatable object which has two different versions (as pointed out, sharing the same data). Specifically, by using the rules in the first `<Executable>` tag (which generates the first version, associated with the `memtrack` suffix), Hijacker scans the whole simulation model's code to find instructions belonging to the `I_MEMWR` family, namely assembly instructions which have a memory address as the destination operand. Among the various ones, the most significant instructions for the x86 architecture (which represents our target) are `mov`, `movs`, and `cmov` instructions. These different instructions are handled internally in a different way by Hijacker, nevertheless before each of them (in the whole original program's image) a call to a specific internal trampoline is placed, along with some instructions which generate an invocation context for it, therefore allowing to identify the characteristics of the original instruction which caused the invocation. This is done via the `<AddCall>` tag, with the `arguments` attribute set to `target`.

The goal of this instrumentation rule is to let Hijacker generate a cache of disassembly information, which can be used at runtime. This step is important to address the efficiency of the runtime execution of our tracking scheme, as it allows us to avoid costly disassembly of instructions at runtime. More specifically, Hijacker extracts from the memory-write instruction the information related to the *size* of the memory write, and the *destination address*. 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, Hijacker packs this information retrieved at compile time in a structure named `insn_entry` which will be directly placed on the stack at runtime, along the execution flow of the running thread. This operation is done by injecting a set of ad-hoc `mov` instructions in the original executable, right before the memory-writing instruction which caused the activation of this procedure, as depicted in Figure 3. To create space for this packed data, Hijacker places as well a `sub` instruction before the `mov` instructions, and a related `add` instruction after the original memory-write instruction, which operate on the stack-pointer. This is to retrieve, during the execution of the instrumented executable, the parameters required to reconstruct the target address (and the size) of the memory update. The `insn_entry` structure is composed of the following fields:

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

where `flags` tells which are the relevant fields of `insn_entry` 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[1].

Additionally to this information, Hijacker places (again, using a couple of ad-hoc `mov` instructions) the address of the function specified in the `function` attribute of the `<AddCall>` tag, `reverse_generator` in our case. This function is defined as `reverse_generator(void *address, size_t size)`, and represents our entry point in the state recoverability manager to build the actual negative instruction which will undo the effect of the memory-write instruction on the simulation state. Since this function could not be present in the original executable[2], Hijacker simply creates a *relocation entry* in the final relocatable object file, leaving to the final linker the task of identifying the correct address.

As mentioned, the actual value for `address` in the most general case can be retrieved only at runtime, as it may depend on the content of the base and/or index registers. Therefore, Hijacker, after having placed the `mov` instructions which put on stack the `insn_info` record associated with the information concerning the current memory-write instruction, inserts a `call` instruction to an internally-defined trampoline which is used to compute the final address. This

[1]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.

[2]This is exactly our case, in fact the module is part of the state recoverability manager, not of the simulation model, which is transparently injected in the original model's code via the `<Inject>` tag specified in the xml-based rules.

```
<hijacker:Rules xmlns:hijacker="http://www.dis.uniroma1.it/~hpdcs/">

        <hijacker:Inject file="mixed-state-saving.c" />

        <hijacker:Executable suffix="memtrack"> <!-- First code version -->

                <hijacker:Instruction type="I_MEMWR">
                        <hijacker:AddCall where="before" function="reverse_generator" arguments="target" />
                </hijacker:Instruction>

        </hijacker:Executable>

        <hijacker:Executable suffix="notrack"> <!-- Second code version -->
        </hijacker:Executable>
</hijacker:Rules>
```

**Figure 2: Hijacker rules to instrument application-level code**

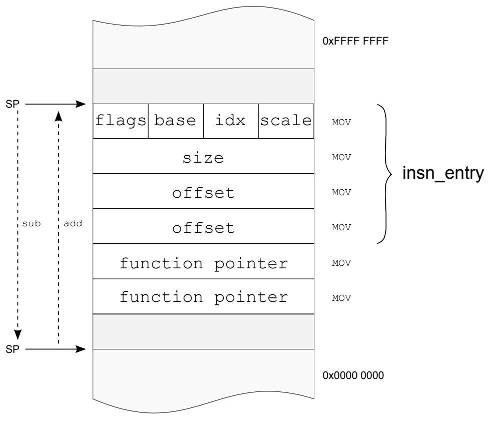

**Figure 3: Trampoline call stack frame**

trampoline exploits the `flags` field of the `insn_info` structure to determine which, among the four parameters *base address*, *index*, *scale*, and *displacement*, determine the memory write address. After having determined this, the trampoline computes the final address and places it either on the stack or in the `rdi`, `rsi` registers, depending on the calling convention of the system (namely, 32-bits vs 64-bits x86). At this point, it retrieves from stack the 4 or 8 bytes composing the address of the function (`reverse_generator` in our case), places it in a general purpose register and performs an indirect call.

To ensure the correctness of the overall original executable, the trampoline is used as well to save the CPU context of the application before executing the final function, and then restores it before giving control back to the original software. Nevertheless, due to the fact that a `sub`/`add` couple of instructions is placed around the original memory-write instruction, Hijacker must as well save the status register of the CPU, namely the `flags` register, in order to leave the flow of execution untouched, so as to let the software continue its execution as if no additional operation was performed. To this end, we place a couple of `pushfw`/`popfw` instructions before and after all the operations on the stack, just to save and restore the status register.

The other two aforementioned types of memory-write instructions have been dealt with in two different ways. On the one hand, the `cmov` instruction is managed directly by the Hijacker's trampoline. Specifically, in case the instrumentation is triggered by a `cmov` instruction, we use 4 bits in the `flags` field of `insn_info` to record what is the actual check to be emulated in order to determine if the memory update will be executed or not. Specifically, 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, by the above discussion, the values of status bits might have been already altered during the execution of the previous injected operations (namely, before the trampoline takes control). To this end, the trampoline's code looks on the application stack for the old value, as stored by the previously-executed `pushfw` instruction. In case the values of these bits, according to the code stored in the `flags` field, tell that the memory-update instruction will be executed, then the control is passed to the `reverse_generator` function, exactly as in the previous scenario. In the negative case, the control is simply returned back to the application. This check is performed right after the trampoline has taken control, in order to avoid the cost of computing the target address in case this information is not useful, thus trying to reduce the cost of this operation.

Concerning the `movs` instruction, 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`.

In all cases, the trampoline is able to compute the tuple $\langle address, size \rangle$, which is passed to the function specified in the xml configuration file (`reverse_generator` in our case). In this way, Hijacker is able (by using the internal trampoline module) to hide the complexity of the underlying hardware architecture and pass the relevant information to the module which will perform the generation of the update undo code blocks. We emphasize that the trampoline has been developed directly in x86 assembly, in a very efficient way, so as to reduce as much as possible the overhead to compute the target address. Additionally, using cached disassembly information allows the trampoline to access all the relevant information very quickly, avoiding the need for costly runtime disassembly, which is more proper of dynamic instrumentation approaches. Moreover, this instrumentation technique allows us to keep, in the same executable, two different versions of the original code, one instrumented to track memory accesses, and one which simply executes the original code. Since by using Hijacker we are able to differentiate these versions by the name of all the functions (namely, one version has all function names with the `memtrack` suffix, the other with the `notrack` suffix), we are able to give control to the two versions of the code very easily. This facility will be exploited later, when this architecture will be coupled with the analytic model to decide which support for the rollback operations should be activated on a very fine grain (e.g. per event) basis.

## 3.2 Runtime Generation of Update Undo Code Blocks

The instrumentation architecture described so far allows, at runtime, to activate the `reverse_generator(void *address, size_t size)` API just before any memory-update operation is performed. At this point, the state recoverability manager is notified of the application code's will to update the simulation model state, and therefore negative instructions (to restore the state in case of a rollback operation) can be built on-the-fly.

Specifically, in case the invocation of `reverse_generator` is related to the execution (in the forward event) of a `mov` or a `cmov` instruction, the negative instruction is simply built by accessing memory at `address` and by reading `size` bytes. This value, since the invocation of `reverse_generator` happens right before the execution of the original memory-write instruction, allows the module to retrieve the "old" value of the simulation model's state. This value is then placed within a data movement instruction as the source (immediate) operand. Of course, the destination of this negative instruction is `address`, as the ultimate goal of this operation is to restore the state by placing back at the same addresses the originally-kept value.

If the activation of `reverse_generation` is due to a `movs` instruction, this can be easily determined by `size`, as it could be higher than the largest representable immediate[3]. In this case, the negative instruction could be only another `movs` instruction. Therefore, `reverse_generator` allocates a memory buffer and creates a copy of the memory pointed by `address`. Then, a set of instructions which place back the same memory content is generated. Of course, this operation is more costly than the generation of a single negative `mov` instruction, but is nonetheless necessary to allow for the correct restoration of the simulation state.

We emphasize that the generation of negative instructions by `reverse_generator` is not a costly operation, except for the `movs` case where a memory buffer must be explicitly copied. In fact, since the set of instructions to be generated is very limited (namely, some variants of `mov` instructions), the opcodes are known beforehand. This allows us to use pre-compiled tables of instructions, where only the relevant parameters should be packed within, namely the old memory value and the destination memory address. With this approach, we are paying an instrumentation overhead which is similar to that of incremental state saving solutions (see, e.g., [19]), but we are completely avoiding any generation of

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

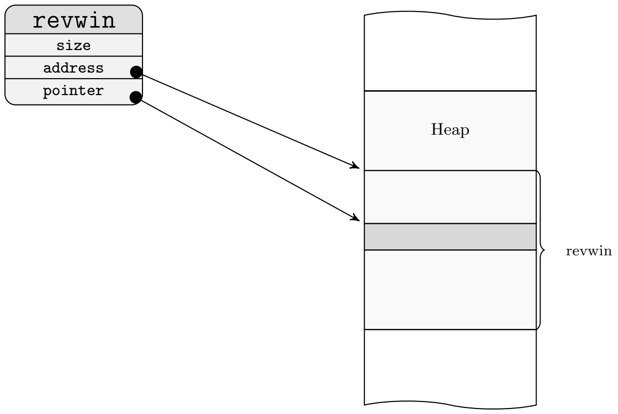

**Figure 4: Revwin descriptor**

metadata, thus reducing the overhead for the installation of a previous snapshot during the execution of a rollback.

The runtime generation module of update undo code blocks offers an additional API, namely `initialize_event(int LP)` which is used to let the simulation engine inform that a new simulation event (during the forward execution phase) is about to be delivered to some LP[4]. Whenever the `initialize_event` API is called, the reverse code generator module allocates on the heap a private region space to dynamically store generated instructions. These instructions are packed into the *reverse window* structure, which is depicted in Figure 4. In this way, both multi-process and multi-thread simulation engines can rely on the reverse generator module, as each LP (and therefore any possible thread executing the LP's code) has its own (private) reverse window, thus easily supporting re-entrant execution. When the reverse window is created, the reverse generator module places at the end of this space a `ret` instruction. This allows to let the execution of the undo event return control to the caller function after the end of the reverse window has been reached. Additionally, the `pointer` field of the `revwin` structure is set to the address of this `ret` instruction.

Each time the `reverse_generator` module is called, a new negative instruction is generated, which is inserted right before the address of `pointer`, whose value is then updated accordingly. In this way, negative instructions are placed within a reverse window (which is associated with one specific simulation event) in the reverse order with respect to the forward execution, which is a fundamental prerequisite for building correct update undo code blocks. In fact, this allows to undo all the effects of a forward event by simply issuing a `call` to the first instruction in the reverse window (i.e., the one pointed by `pointer`).

In case one event requires the generation of a large number of negative instructions, the reverse window's space could get exhausted. In this case, the reverse generator module doubles the size of the reverse window. This operation could be costly, as it requires shifting the content of the current reverse window (namely, all the negative instructions) to the end of the newly allocated memory area, and updating `pointer` accordingly. To reduce the frequency of this opera-

[4]In our architecture, each LP is associated with a unique numerical ID in the range $[0, N-1]$. Any simulation engine using a different identification strategy can, nevertheless, rely on some mapping function to integer values.

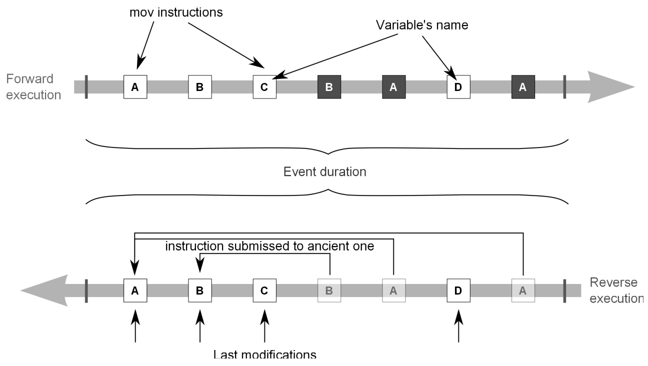

**Figure 5: Instruction predominance**

tion, the initial size of a reverse window can be specified at compile time according to conservative (size overestimation) approaches.

To reduce the amount of negative instructions which are stored into a reverse window, we have explicitly addressed the case where, during the execution of an event in forward mode, the same memory location is touched multiple times in write mode. Due to the impulsive nature of discrete events, these multiple updates should be all undone during a rollback operation. Therefore, generating multiple negative instruction for a same memory target into the update undo code blocks would be both a waste of space in the reverse window and an additional non-necessary cost when executing the actual update undo instructions. This scenario is depicted in Figure 5.

We therefore employ an ad-hoc data structure to keep track of referenced addresses, namely a fast hashmap. Whenever `reverse_generator` is activated, this hashmap is queried to determine whether (within the same event's execution) the destination address was already involved in a negative instruction generation. Basically, this hashmap exploits a two-level bitmap to coalesce multiple addresses within a single word, so as to optimize space requirement for address mapping. A toggle bit is used to indicate if an address is already referenced by some memory-write instruction or not. The structure is a linear array of elements treated as a bidimensional matrix. Each element of the array is a quadword of 64 bits used as basic storage unit for a single range of family's addresses[5]. To access the map, the following two values are needed: (i) an *index* providing the address family range, and (ii) the *offset* which identifies the address' bit within the storage unit (i.e. the quadword). These are computed by properly masking the address value. A family range is therefore composed by all the addresses whose value starts with the same prefix. The length of this prefix depends on the number of flags the storage unit can contain, namely a quadword in our case. This can store up to 64 flags. Given the address, the *offset* is computed by extracting the least $n-1$ significant bits, while *index* is computed as the result of a bitwise-AND with the remainder of most significant bits. Figure 6 shows an example of address' binding for a 32-bit architecture.

To correctly keep a per-event reverse window, whenever the simulation engine invokes the `initialize_event` API,

[5]This allows us to handle both 64-bits x86 architectures and 32-bits ones, at the cost of wasting some space if running on older CPUs.

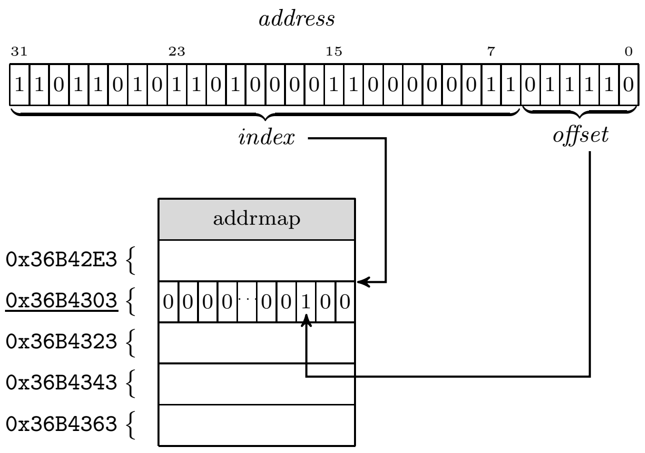

**Figure 6: The `index` and `offset` bitmasks of revwin's hashmap**

the hashmap's content is flushed, so as to allow the new reverse window to store all the negative instructions, even though previously-executed events touched the same memory addresses.

## 3.3 Model-based Optimization of the Recoverability Support

As for models aimed at optimizing the parameters driving the execution of recoverability (and actual recovery) tasks in optimistic PDES, the one presented in [7] can be used as a basis for optimizing the configuration of the specific recoverability support we are presenting. Particularly, in this work the authors consider a scenario where full and incremental logs (in the form of before images) are mixed in a same checkpoint interval. A full log is taken each $\chi$ event executions, and then the LP executes a number $\mu$ of events without saving before-images of updated memory locations, then for the remaining $\nu = \chi - 1 - \mu$ events in the same checkpoint interval, before-images of the state variables are logged. In this way a state can be recovered by reloading an older checkpoint and the coasting forward, or by reloading a later checkpoint and applying the incremental changes in backward mode. A final equation is achieved that describes the trade-off between recoverability tasks and actual recovery, namely:

$$\frac{(\delta_s + \nu\delta_{bi})}{\chi} + F_r \left[\frac{\chi-\nu}{\chi}\left(\delta_r + \frac{\chi-\nu-1}{2}\delta_e\right) + \frac{\nu}{\chi}\left(\delta_r + \frac{\nu}{2}\delta_b\right)\right]$$

where:

- $\delta_s$ is the average time to take a full log of the LP state;
- $\delta_{bi}$ is the average time for saving the before-images of the state variables during the execution of an event;
- $\delta_r$ is the average time to reload a full checkpoint from the log;
- $\delta_e$ is the base execution time of the event (not including the cost for saving before images in case of memory updates);
- $\delta_b$ is the latency for one backward recovery operation that reloads the before images for a specific event execution to be undone;

$F_r$ is the frequency of rollback of the LP (classically evaluated as the number of rollbacks over the total number of event executions).

The minimization of this equation vs the tunable parameters $\chi$ and $\nu$ (as depicted in [7]) leads to the optimal combination of coasting forward based recovery, and backward recovery. This same equation can be applied to our architecture when considering that $\delta_{bi}$ will correspond to the time for running the injected code that traps memory accesses and builds (packs) the update undo code blocks, and $\delta_b$ is the average time for running the undo update code block associated with a specific event.

Let us again stress that reusing the above model for optimization purposes of the configuration of the recoverability support based on update undo code blocks does not reduce the level of innovation by our proposal, given that the original solution in [7] had no support for application level transparency, rather all the checkpoint operations (incremental and non-incremental) were demanded to the application level code. Full application transparency is instead guaranteed in our approach.

### 3.4 Final Integration

Once the execution of a simulation event is completed, the reverse code generation module ensures that all the relevant negative instructions have already been assembled in the current reverse window. We have therefore augmented the set of API functions offered by the reverse code generation module including `void *get_last_window()`, which allows to retrieve a pointer to the just packed `revwin` data structure. This pointer can be therefore stored by the simulation engine directly in the event queue (specifically, in the just processed event node), so as to allow for a fast retrieve in case the event should be undone due to a rollback operation.

Once the simulation engine detects that an out-of-order event $e$ associated with timestamp $T_e$ has been received, the rollback operation is actually executed according to the following algorithmic steps:

1. The event $e$ is incorporated into the event queue of the destination LP. The next event $e_{next}$ in the queue if found;
2. If this event has a pointer to a reverse window, then by the actual operating mode of the recoverability support that combines coasting forward and update undo code blocks (namely, the one whose analytic model for its optimization has been discussed in Section 3.3) we can execute a reverse reconstruction. If the pointer is not set, then a traditional state restore (possibly involving a coasting forward operation) is executed;
3. If a reverse window is present, then the log queue is scanned in order to find a checkpoint $C_{next}$ associated with timestamp $T_{C_{next}}$ such that $T_{e_{next}} \leq T_{C_{next}}$, if any;
4. If this checkpoint does not exist, then it means that we are rolling back to a recent simulation time, so we can simply start executing update undo code blocks. If $C_{next}$ is found, then the simulation engine restores this state, and sets $e_{rev}$ to the last event executed before $C_{next}$ was taken;
5. All update undo code blocks between $e_{rev}$ and $e$ are executed in reverse order.

The execution of an update undo code block is left to the state recoverability architecture, via the `undo_event(void *)` API function. This function takes a pointer to a reverse window, determines the value of the associated `pointer` fields and issues an indirect call to that address. The presence of a `ret` instruction at the end of the reverse window ensures that, after having undone the event, control is given back to the rollback algorithm of the simulation engine. A similar procedure as the one depicted above is executed in case of a rollback caused by an anti-message.

To recover memory, the logic associated with the traditional fossil collection operation should be augmented. Specifically, during the fossil collection, when the input queue of some LP is pruned from older events, the simulation engine is offered the `release_window(void *)` API function, which allows to release all the memory used for the update undo instructions of one event. In this way, we are able to recollect memory, which can be used again during forward execution to maintain reverse instructions related to the execution of additional simulation events.

## 4. EXPERIMENTAL RESULTS

### 4.1 Test-bed Environment

We have integrated the presented state recoverability manager within the ROOT-Sim simulation platform [18, 20]. 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. More in detail, we have integrated our innovative state recoverability support in the symmetric multi-threaded version of ROOT-Sim[6] that has been presented in [33].

This platform has 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.

Our integration has been carried out following two steps. On the one hand, we have altered the final executable generation chain proper of ROOT-Sim, adding one step which involves the actual invocation of Hijacker to instrument the application-level code. On the other hand, we have added some logic to the simulation kernel in order to execute the rollback operation adopting our update-undo-based recoverability support.

As for the first step, ROOT-Sim relies on the `rootsim-cc` custom compiler to generate the final simulation model's executable, carrying out several steps in order to correctly link to the set of static libraries proper of the simulation engine. In particular, during the compilation of a simulation model, `rootsim-cc` performs the following steps:

1. All the sources from the model are compiled using the standard `gcc` compiler, and one single relocatable object file is produced.

[6]The source of the multi-threaded version of ROOT-Sim, along with the presented state recoverability support, can be found at `http://github.com/HPDCS/ROOT-Sim`.

2. This relocatable object file is then incrementally linked via `ld` to the DyMeLoR static library. In this process, all the calls to the `malloc` standard library are redirected to the proper DyMeLoR allocator (see [21] for a thorough description of DyMeLoR and this compilation step).
3. Then, the produced incrementally-linked relocatable object is again incrementally linked to an additional static library (called `libwrapper`) which allows for the redirection of all stateless library functions proper of the `C` standard library to a set of wrappers which allow for a correct integration with the DyMeLoR library.
4. Finally, this new relocatable object is linked to the final `librootsim` library.

We have altered this compilation process by inserting an additional step right after Step 1 in the previous list. In particular, during this additional step we explicitly call Hijacker, passing the configuration-rule set shown in Figure 2. This allows us, as discussed, to generate a model relocatable object file which is already integrated with the innovative state recoverability support.

As a note, we emphasize that calling Hijacker before linking the model with DyMeLoR does not pose any issue regarding memory management, as the reverse generation module relies on `mmap` to allocate memory to keep reverse instructions, which is not redirected by DyMeLoR itself, and therefore allows to keep these buffers away from the LPs' simulation state.

## 4.2 Test-Bed Application Model

We have run experiments to assess the overall performance and behavior of the presented state recoverability architecture by relying on the Personal Communication System (PCS) benchmark, which models a mobile network adhering to GSM technology. Each LP models the state's evolution of an individual hexagonal cell, and the whole set of cells provides wireless coverage on a square region of variable size. Each cell handles a parameterizable number $N$ of wireless channels, which are modeled in a high fidelity fashion via explicit simulation of power regulation and interference/fading phenomena, according to the results in [13].

The event types which can occur at any LP are: *Start Call*, which simulates a new call installation on a target cell; *End Call* which simulates a call termination; *Handoff Leave* which simulates the leave of an on-going call from the current residence cell; *Handoff Receive* which simulates the installation of a call handed off from an adjacent cell; *Recompute Fading*, which simulates the effects of weather variations onto the fading (and consequently interference) phenomena for ongoing calls.

Upon the start of a call, a call-setup record is instantiated via dynamically-allocated data structures, which is linked to a list of already active records within that same cell. Each record is released when the corresponding call ends or is handed off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call setup to achieve the threshold-level Signal-to-Interference Ratio (SIR) value. Data structures keeping track of fading coefficients are also updated while scanning the list, and on a periodic basis, according to a

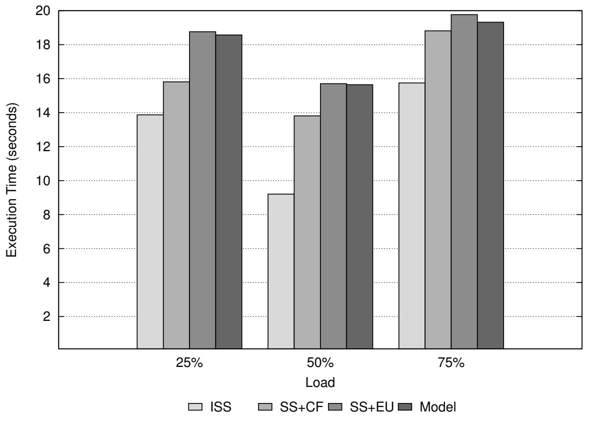

**Figure 7: PCS execution time with 64 LPs**

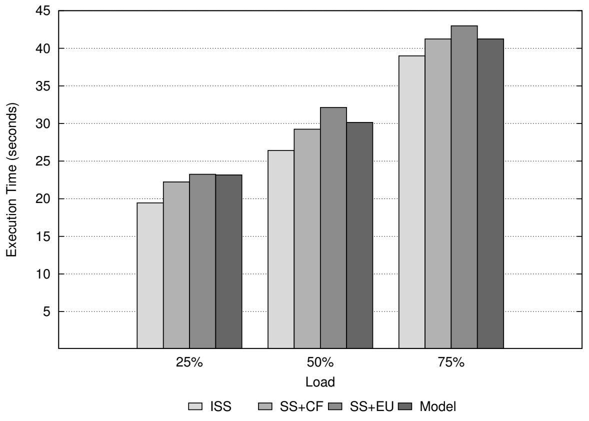

**Figure 8: PCS execution time with 256 LPs**

meteorological model defining climatic conditions (and related variations).

This application is highly parameterizable. Beyond the already mentioned number $N$ of wireless channels per cell, the set of configurable parameters entails: (i) $\tau_A$, which expresses the inter-arrival time of subsequent calls to any target cell; (ii) $\tau_{duration}$, which expresses the expected call duration; (iii) $\tau_{change}$, which expresses the residual residence time of a mobile device into the current cell. These parameters affect the *utilization factor* of available channels, expressed as $\tau_{duration}/(\tau_A * N)$. This impacts the granularity of the events since the more channels are busy, the more power-management records are allocated and consequently scanned/updated during the processing of different events. On the other hand, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual LPs. Both the above dependencies (namely, CPU demand and memory) are anyhow bounded depending on the total number $N$ of per-cell managed channels.

## 4.3 Experimental Data

To study the effects of our state recoverability architecture when considering differentiated execution and memory access patterns for the application layer, we have run experiments varying the actual size of the model, setting the number of simulated cells (and thus, of LPs) in the interval

**Table 1: Sequential execution time (seconds)**

| Load | 64 LP | 256 LP | 576 LP | 1024 LP |
|---|---|---|---|---|
| 25% | 68.249 | 188.821 | 412.830 | 847.429 |
| 50% | 80.256 | 310.822 | 716.359 | 1266.089 |
| 75% | 117.117 | 433.494 | 1058.123 | 1772.595 |

[64, 1024]. Each cell handles 1000 wireless channels, and we have varied the call inter-arrival frequency so as to obtain an average utilization factor in between 25% and 75%. We have run the simulation models until each cell has managed 10000 completed calls, assessing the overall execution time (and therefore the performance) of different configurations of the simulation engine. Specifically, we compare the traditional rollback technique based on state saving and coasting forward (referred to as SS+CF), the rollback technique relying on incremental state saving as presented in [21] (referred to as ISS), the rollback technique based on the execution of our update undo code blocks (referred to as SS+EU) and the interleave of SS+CF and SS+EU based on the analytic model presented in [7]. For completeness, we also report the execution times for the case of a serial execution of the same identical configurations on top of a calendar queue-based sequential scheduler [5], to confirm that this study refers to competitive parallel performance. All the results are averaged over 5 different runs, configured with the same initial pseudo-random seeds used by the random number generators. All experiments have been carried out using 32 concurrent worker threads in the simulation engine, and in all configurations the checkpointing interval $\chi$ has been set to 10, so as to allow for an even comparison between the differentiated approaches (i.e., there is no actual benefit by the checkpointing interval for any of the presented configurations). In fact, since the various techniques that we are comparing in this study can rely on different approaches to fine tune $\chi$ (let them be iterative, analytic, hill-climbing-based, ...), we have decided to set this parameter to a fixed value, so as not to create any bias favoring any approach. Given this selection, the model based approach presented in [7], which we have exploited in Section 3.3, has been used to derive the optimal value $\nu$, starting from this fixed value of $\chi$.

In Figures 7–10, we present the overall execution time of our simulations in the aforementioned configurations. We can see that the parallel runs offer a speedup with respect to the sequential execution (as reported in Table 1) ranging from 5 (in case of 64 LPs) to 14 (in case of 1024 LPs). Therefore, this experimentation allows us to assess the behavior of our proposed architecture in scenarios with a high degree of parallelism and a somewhat limited efficiency of the parallel run, and scenarios where the efficiency is much higher.

By the results, we can see that when the number of LPs is limited (Figures 7 and 8), independently of the load of the system the best performance is provided by ISS. This is related to the fact that, although the operations required by ISS to mark a portion of the simulation state as modified require scanning complex metadata, all the tables are already allocated in memory. On the contrary, in the case of SS+EU each event requires allocating a new reverse window. When the load is limited (namely, 25%), by the configuration of our benchmark application, the amount of write operations

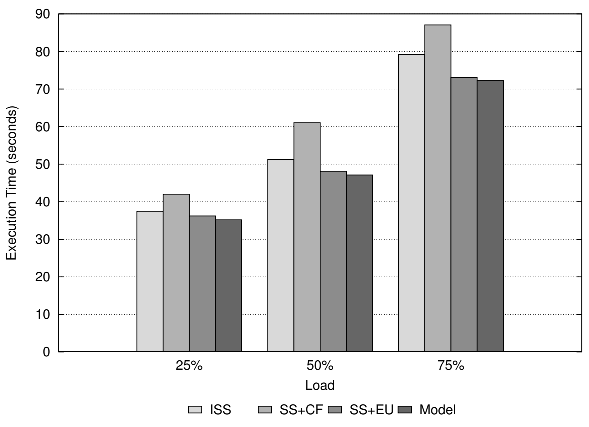

**Figure 9: PCS execution time with 576 LPs**

per each event is reduced. Therefore, the cost of allocating the reverse window and the cost of generating negative instructions at runtime are not amortized by the events' duration. Furthermore, concerning the rollback operation, the amount of data touched in write mode by the model with a low load is reduced. Thus, this configuration can be regarded as mostly an assessment of the overhead produced by our state recoverability architecture. Overall, the slowdown of SS+EU with respect to ISS is on the order of 30%, while with respect to SS+CF is on the order of 15%. Conversely, when the load of the system is higher, namely 75%, the amount of data touched in write mode is higher. Therefore, the ratio between the number of created reverse windows and the amount of generated negative instructions is lower. This reduces the relative overhead by our state recoverability architecture, due to a better exploitation of data locality in the undo log (namely, the update undo code block) by the rollback operation. In this configuration, the maximum slowdown by SS+EU with respect to ISS is on the order of 15%. In all the configurations, the model-based scheme allows to reduce the overhead by SS+EU, with a performance increase on the order of 5%.

When the number of LPs involved in the simulation is larger (namely, as reported in Figures 9 and 10), the situation changes. When the workload is reduced, SS+EU provides a performance increase on the order of 5% with respect to ISS, and on the order of 25% with respect to SS+CF. This is mainly related to the fact that this scenario offers a lower degree of parallelism, and thus the number of rollback operations is reduced. Therefore, this configuration allows us to assess that the overhead to generate a log by ISS and SS+CF is higher than the overhead by SS+EU to generate the update undo code blocks. When the workload increases, namely it reaches 75%, this phenomenon is exacerbated. In fact, similarly to the configurations with 64 and 256 LPs, the number of negative instructions per undo event is higher, which compensates for the cost of allocating memory and setting up the associated data structures. In this context, the performance gain of SS+EU with respect to ISS is on the order of 10%, while it is on the order of 20% with respect to SS+CF. Analogously to the configurations with 64 and 256 LPs, the model-based approach allows for an even higher performance increase, up to 32%.

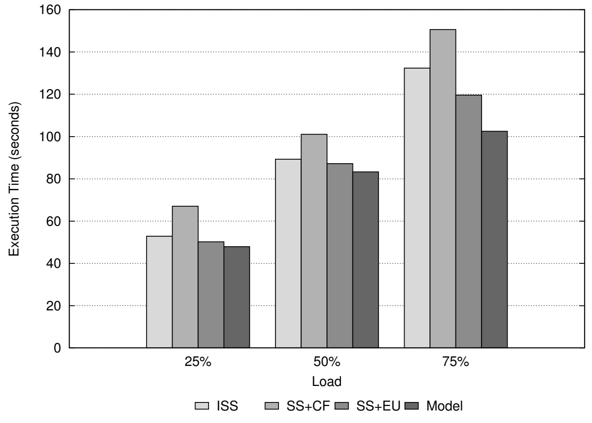

**Figure 10: PCS execution time with 1024 LPs**

Overall, the data show our state recoverability solution to be the most effective one in larger configurations of the test-bed application, exhibiting heavier workloads, which are, after all, the key scenarios where (optimistic) parallel simulation is considered to be of wide usage.

## 5. CONCLUSIONS

In this paper we have presented the design and implementation of a practical state recoverability manager for optimistic PDES that mixes undo logs and software reversibility techniques. Our proposal is completely transparent to the application-level developer, by relying on static binary instrumentation to detect what are the assembly instructions which perform memory updates during the execution of an event in the speculative forward phase. By exploiting this information, we have proposed an approach to generate at runtime update-undo code blocks, which are able, when executed, to revert the effects of the execution of an event on the LP's state. We have shown this solution, as well, to lie as a practical support for the transparent implementation of decision models, which can be used to determine at runtime what is the most effective (forward) execution mode, in terms of its recoverability support.

Our experimental assessment highlights that relying on update undo code blocks to put in place rollback operations can provide benefits on execution timeliness, when compared to other traditional supports.

## 6. REFERENCES

[1] GDB: The GNU Project Debugger. *http://www.gnu.org/software/gdb/* (last accessed: May 11th, 2015).

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

[3] P. D. Barnes, Jr., 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* (SIGSIM-PADS), pages 327–336. ACM Press. Montréal, Canada – May 19–22, 2013.

[4] 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. Society for Computer Simulation International. Newport Beach, California – January 20–22, 1992.

[5] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[6] 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.

[7] V. Cortellessa and F. Quaglia. A checkpointing–recovery scheme for Time Warp parallel simulation. *Parallel Computing*, 27(9):1226–1252, 2000.

[8] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In *Proceedings of the Winter Simulation Conference* (WSC), pages 1332–1339. Society for Computer Simulation International. Orlando, FL, USA – December 11–14, 1994.

[9] 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* (PADS), pages 50–58. IEEE Computer Society. Lake Placid, NY, USA – June 13–16, 1995.

[10] S. Franks, F. Gomes, B. Unger, and J. Cleary. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation* (PADS), pages 72–79. IEEE Computer Society. Lockenhaus, Austria – June 10–13, 1997.

[11] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

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

[13] S. Kandukuri and S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[14] 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* (SIGSIM-PADS), pages 3–14. ACM Press. Denver, CO, USA – May 18–21, 2014.

[15] Y.-B. Lin and E. D. Lazowska. Reducing the state saving overhead for Time Warp parallel simulation. *Tech. Rep. 90-02-03, Department of Computer Science and Engineering, University of Washington, Seattle, Feb. 1990.*

[16] 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. San Diego, CA, USA – May 16–19, 1993.

[17] A. Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing. In *Proceedings of the 2013 International*

*Conference on High Performance Computing & Simulation* (HPCS), pages 650–655. IEEE Computer Society. Helsinki, Finland – July 1–5, 2013.

[18] A. Pellegrini and F. Quaglia. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations* (PADABS). LNCS, Springer-Verlag Berlin Heidelberg. Aachen, Germany – August 26–30, 2013.

[19] 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 of the 23rd Workshop on Principles of Advanced and Distributed Simulation* (PADS), pages 45–53. IEEE Computer Society. Lake Placid, NY, USA – June 22–25, 2009.

[20] A. Pellegrini, R. Vitali, and F. Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques* (SIMUTools), pages 96–98. ICST. Barcelona, Spain – March 22–24, 2011.

[21] A. Pellegrini, R. Vitali, and F. Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems (preprint available)*, May 2014. doi: 10.1109/TPDS.2014.2323967.

[22] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, 1994.

[23] 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 39th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 135–148. IEEE Computer Society. Orlando, FL, USA – December 9–13, 2006.

[24] 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.

[25] F. Quaglia and A. Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, 2003.

[26] R. Rönngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (PADS), pages 110–117. Society for Computer Simulation. Edinburgh, Scotland – July 6–8, 1994.

[27] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (PADS), pages 70–77. IEEE Computer Society. Philadelphia, PA, USA – May 22–24, 1996.

[28] S. Skold and R. Rönngren. Event sensitive state saving in Time Warp parallel discrete event simulation. In *Proceedings of the Winter Simulation Conference* (WSC), pages 653–660. Society for Computer Simulation. Coronado, CA, USA – December 8–11, 1996.

[29] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, 1998.

[30] J. S. Steinman. SPEEDES—a multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation*, 2:251–286, 1992.

[31] J. S. Steinman. Incremental state saving in SPEEDES using C plus plus. In *Proceedings of the Winter Simulation Conference* (WSC), pages 687–696. Society for Computer Simulation. Los Angeles, CA, USA – December 12–15, 1993.

[32] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation* (PADS), pages 163–172. IEEE Computer Society. Rome, Italy – June 3–6, 2008.

[33] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation* (PADS), pages 211–220. IEEE Computer Society. Zhangjiajie, China – July 15–19, 2012.

[34] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 1–12. ACM Press. Albuquerque, NM, USA – June 21–25, 1993 .

[35] 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. Philadelphia, PA, USA – May 22–24, 1996.

[36] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In L. Hendren, editor, *Compiler Construction*, LNCS, Springer-Verlag Berlin Heidelberg. Budapest, Hungary – March 29 - April 6, 2008.