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

DAVIDE CINGOLANI, Sapienza University of Rome
ALESSANDRO PELLEGRINI, Sapienza University of Rome
FRANCESCO QUAGLIA, Sapienza University of Rome

The Time Warp synchronization protocol for Parallel Discrete Event Simulation (PDES) is universally considered as a viable solution to exploit the intrinsic simulation model parallelism and to provide model execution speedup. Yet, it leads the PDES system to execute events in an order which may generate causal inconsistencies that need to be recovered via *rollback*, which requires restoration of a previous (consistent) simulation state any time a causality violation is detected. The rollback operation is so critical for the performance of a Time Warp system that it has been extensively studied in the literature for decades, to find approaches suitable to optimize it. The proposed solutions can be roughly classified as based on either *checkpointing* or *reverse computing*. In this article, we explore the practical design and implementation of a fully new approach based on the runtime generation of so called *undo code blocks*, which are blocks of instructions implementing the reverse memory side-effects generated by the forward execution of the events. However, this is not done by recomputing the original values to be restored, as instead it occurs in reverse computing schemes. Hence, the philosophy undo code blocks rely on is similar in spirit to that of undo-logs (as a form of checkpointing). Nevertheless, they are not data logs (as instead checkpoints are), rather they are logs of instructions. Our proposal is fully-transparent, thanks to the reliance on static software instrumentation (targeting the x86 architecture and Linux systems). Also, as we show, it can be combined with classical checkpointing, so as to further improve the runtime behavior of the state recoverability support as a function of the workload. We also present experimental results related to our implementation, which is released as free software and fully integrated into the open source ROOT-Sim (The ROme OpTimistic Simulator) package. Experimental data support the viability and effectiveness of our proposal.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code Generation*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*; *parallel*

General Terms: Theory, Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Parallel Discrete Event Simulation, Optimistic synchronization, Time Warp, Software Reversibility

1. INTRODUCTION

Discrete Event Simulation (DES) is a methodology that models the behavior of a given system using a sequence of events, and associated state transitions, occurring at discrete points in time [Robinson 2004]. Parallel Discrete Event Simulation (PDES) [Fujimoto 1990] is the set of techniques aimed at exploiting hardware parallelism for speed-

Authors' addresses:
D. Cingolani—Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti" – Sapienza, University of Rome – Via Ariosto 25, 00185 Roma, Italy, email cingolani@dis.uniroma1.it;
A. Pellegrini—Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti" – Sapienza, University of Rome – Via Ariosto 25, 00185 Roma, Italy, email pellegrini@dis.uniroma1.it;
F. Quaglia—Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti" – Sapienza, University of Rome – Via Ariosto 25, 00185 Roma, Italy, email quaglia@dis.uniroma1.it

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ing up the execution of DES models. In this area, Time Warp [Jefferson 1985] is one of the fundamental PDES synchronization protocols proposed in literature.

According to classical PDES, in Time Warp the simulation model is partitioned into distinct simulation objects, which are mapped to Logical Processes (LPs). They handle the execution of the discrete events, which ultimately produce updates to the actual simulation model's state, and which may schedule other events to occur in the future. Given its speculative nature, also referred to as *optimism*, Time Warp leads the LPs to process events independently of their safety (or causal consistency). Therefore, there is a potential for great exploitation of the intrinsic model's parallelism. If some processed event is a-posteriori detected to be violating causality (i.e., it has been processed out of timestamp order), its effects on the simulation state are undone via the *rollback operation*. Correctly and efficiently rolling back the states of the LPs is therefore a fundamental building block for an effective Time Warp system. In the literature, this operation has been thoroughly studied, and different approaches have been proposed, which can be roughly grouped into two separate families: *checkpoint-based* [Jefferson 1985] and *reverse computing-based* [Carothers et al. 1999], depending on the technique that is used to bring the LP state to a previous (consistent) snapshot.

The checkpoint-based rollback operation requires the PDES engine to know the memory location of each part of the state of an LP. By exploiting this information, the engine can create a copy of the state before the execution of operations leading to state updates. In this context, different solutions have been presented, all aimed at reducing the cost, in terms of memory and CPU usage, paid to create a state snapshot, which will be (possibly) used for a later restore operation. Among the various research lines, we find two main approaches. On the one hand, we find solutions to reduce the frequency according to which the snapshots of the LP state are taken—the so-called *sparse* or *periodic state saving* [Lin and Lazowska 1990; Bellenot 1992; Palaniswamy and Wilsey 1993; Preiss et al. 1994; Rönngren and Ayani 1994; Fleischmann and Wilsey 1995; Skold and Rönngren 1996; Quaglia 2001]—with a focus on detecting the best-suited checkpointing interval to minimize unfruitful work (i.e., taking checkpoints whose overhead does not pay off in reducing state recovery costs). On the other hand, we find solutions which try to reduce the amount of data copied into a state snapshot, ensuring that no meaningful piece of information is lost—the so-called *incremental state saving* [West and Panesar 1996; Pellegrini et al. 2009]. Mixtures of these approaches have been proposed as well [Cortellessa and Quaglia 2001; Pellegrini et al. 2015], which modify (possibly at runtime) the execution mode of the state saving operation, depending on the execution dynamics of the overlying model. Further, approaches based on offloading checkpoint tasks from the CPU have also been in investigated [Fujimoto et al. 1992; Quaglia and Santoro 2003].

The reverse computing-based rollback operation is based instead on the availability of *reverse event handlers*, such that if the forward execution of an event e on an LP state S produces a state transition $e(S) \rightarrow S'$, then the corresponding reverse handler r would produce on S' the inverse transition $r(S') \rightarrow S$. Reverse event handlers can be provided either manually [Carothers et al. 1999; Seal and Perumalla 2011] or automatically [Hou et al. 2012b; LaPre et al. 2014]. Overall, while the reverse computing approach is able to significantly reduce the impact of memory usage from which checkpointing may suffer—checkpoints are no longer the systematic means to achieve state recoverability—the execution cost of the rollback operation is directly proportional to the execution time of simulation events, particularly in reverse mode.

To some extent, all the proposed approaches have some drawback. One above all is the dependency of the recovery cost on the granularity of the events (either their forward mode execution or their reverse mode execution). Particularly, when relying on infrequent checkpointing, the lower the frequency with which the snapshots (in-

cremental or not) are taken, the higher is the cost for restoring a state that is not checkpointed. In fact, the restore operation needs to reload an older snapshot and to reprocess intermediate events. As shown in the literature, the best tradeoff is a function of the ratio between average checkpointing cost and event granularity (see, e.g., [Palaniswamy and Wilsey 1993; Preiss et al. 1994; Rönngren and Ayani 1994; Fleischmann and Wilsey 1995]). On the opposite side, pure reverse computing may suffer from high restore costs when the rollback length is non-minimal, since many backward events would need to be processed as input to reverse event handlers. This is unfavorable especially if the reverse handlers need to implement costly reverse computation steps, which might be the case for complex events.

By mixing the different philosophies behind these recoverability techniques, we present a new approach based on the combination of undo-logs with software reversibility. In our proposal, data typically recorded by undo-log systems (which are essentially checkpointing systems) are used to generate so called *undo code blocks*, which are not undo-logs of data, rather of runtime (dynamically) assembled machine instructions. Undo code blocks can be used to squash memory updates generated by a causally-inconsistent portion of the computation. However, this does not rely on backward re-computation of the values to be restored via reverse event handlers, as it happens in reverse computing. To achieve our goal, we rely on software instrumentation, which allows us to capture the *effects* on memory by the forward execution of the events. This information is then used to build at runtime the undo code blocks, which are compact undo-logs of actual operations. They are compact because they are directly encoded as a set of consecutive machine instructions which cancel the effects of the execution of forward events. As hinted, 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 undo code block is only proportional to the amount of memory areas touched in write mode during forward execution.

As pointed out, our approach is also different from classical undo-logs, which are not based on dynamic generation of reversibility code, rather they only log (and restore) data. Further, our solution is specifically designed to avoid the cost paid by incremental checkpointing systems when executing a rollback operation. In fact, while the latter generally require inspection of some (arbitrarily complex) metadata to determine where each portion of the incremental log should be placed in memory, undo code blocks can be simply executed with no additional overhead associated with metadata management. Indeed, they are specifically generated so as to keep all the relevant information already packed within a block of machine code.

Nevertheless, our proposal can be combined with classical checkpointing. Particularly, by taking infrequent checkpoints, we can generate checkpoint intervals where some passed-through snapshots are not recoverable by undo code blocks, while others are. Hence state restore can be executed by either reloading a previous checkpoint and executing a classical coasting forward phase, or by applying backwards operations encoded within undo code blocks starting from the current state snapshot (or from a conveniently selected checkpoint). While the former case occurs for the restore of a snapshot uncovered by the undo facility, the latter occurs for the restore of covered snapshots. 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.

Borrowing from the results in [Cortellessa and Quaglia 2001], we devise an analytic model and a heuristic scheme for the dynamical re-configuration of recoverability based on undo code blocks. This allows us to determine how to optimize the partitioning of each single checkpoint interval, also depending on factors like the event granularity. The result is an optimization of the the length of the interval, as well as

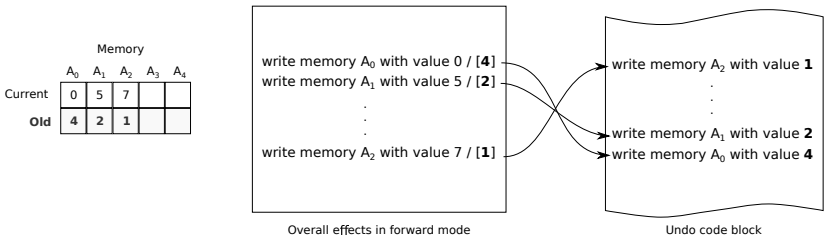


Fig. 1. Relation between memory updates in the forward execution of an event e and the corresponding undo code block. Old values are represented in brackets.

the selection of what are the snapshots to reconstruct in case of a rollback by relying either on coasting forward or undo code blocks.

We also report experimental data for an assessment of our proposal which, we remark, is fully application-transparent and has been released as free software[1] via the integration into the open source ROOT-Sim package [Pellegrini and Quaglia 2013].

The remainder of this article is structured as follows. The undo code block-based recoverability approach is described in Section 2. In Section 3 we present the implementation of the overall recoverability architecture based on undo code blocks. Section 4 is devoted to the experimental assessment. Related work is discussed in Section 5.

2. UNDO CODE BLOCK-BASED STATE RECOVERABILITY

In this section we initially provide the basic concepts underlying our innovative state recoverability proposal. Then we discuss optimizations related to locality of the updates occurring along forward execution of the events at the LPs. Finally, we describe how the undo code block-based technique can be exploited in synergy with classical checkpointing, also illustrating a performance model for tuning the parameters driving the operations of the integrated recoverability support.

2.1. Basics

Any event e speculatively executed at LP_i produces the state transition $e(S_i) \rightarrow S_i'$ on the private state S_i of LP_i. Our approach to undo the processing of e in a rollback phase is based on a reverse transition $r_e(S_i') \rightarrow S_i$ which is only aimed at canceling the effects on memory of the forward execution of e. Hence, reversing memory side effects associated with the transition $e(S_i) \rightarrow S_i'$ is the exact objective of the undo code block to be associated with the event e. We again recall that an undo code block is a runtime-built set of machine instructions.

Every memory update operation executed by any event e during its forward execution must be intercepted, and this needs to be done transparently, with no intervention by the simulation model programmer. As we shall describe in detail in Section 3, we rely on *static binary instrumentation*[2], a technique which modifies at compile time the image of an executable by inserting, according to some rules, additional assembly instructions that are later executed at runtime. These instructions must avoid any change in the *semantics* of the program, namely the final outcome of the execution of an event must be the same as if no additional instructions were injected at all.

The role of the additional instructions is to determine the target address of a memory-update operation *before* it takes place, so that the memory location being mod-

[1] `https://github.com/HPDCS/ROOT-Sim`

[2] We target the x86_64 ISA and the ELF file format on Linux systems, which allows us to cover more than 95% of current high-performance clusters on top500 [Pellegrini 2015]. In any case, the methodology we propose can be easily ported to other operating systems, executable formats and ISAs.

ified can be accessed to read its current value. This value is what the undo code block should restore upon the execution of a rollback operation. Overall, by using static binary instrumentation, the simulation engine can determine at runtime what memory locations are being updated by the execution of an event e during its forward execution phase, and this information is what we exploit to implement the reverse transition $r_e(S'_i) \rightarrow S_i$ to be fired in case of rollback.

In more detail, the simulation engine generates at runtime an assembly instruction which uses the original memory value (namely, the value found at the computed memory address before the update operation takes place) as the *source operand*, and the computed memory address as the *destination operand*. Such an instruction is then placed in a contiguous buffer used as a stack of generated reverse instructions, which are found in reverse order from the top of the stack. Executing them according to such reverse order undoes the effects of memory updates perpetrated by the forward execution of any event e. This scheme is illustrated in Figure 1, where different memory locations are updated when running in forward mode. The corresponding reverse instructions generated at runtime in the undo code block associated with e, which we refer to as ub_e, put back in the correct place the previous values found in memory.

It is interesting to note that the computation in forward mode generating the values used to update the target memory locations can be of any arbitrary complexity. Indeed, the updates performed by the event e along forward execution can result from any kind of algorithm used by the simulation code. Since we are generating ub_e at runtime, we are not interested in *how* this value was computed. We are just interested in determining *what* is the value of the target memory location which is about to become "old." Therefore, since we track memory update operations at runtime right before they take place, the corresponding reverse phase will not entail recomputing the "old" value, since it is already encoded in the dynamically-generated reverse instruction corresponding to the memory update. Overall, a reverse instruction in an undo code block does not depend on the complexity of the algorithm which generates the forward memory update. Hence, it can undo in constant-time the side effect generated by the computation of a value, regardless of the generation complexity.

Anyhow, a single event e executed in forward mode might update multiple times the same memory location. This could be the case of algorithms using simulation state variables as accumulators, where partial results of the computation of an event are temporarily stored, since the computation of the final value requires multiple steps within the same event. In this case, to reduce the amount of negative instructions which must be generated during the forward execution of the event e (and therefore the number of reverse instructions placed within ub_e to be executed in reverse mode in case of rollback) we exploit the *Instruction Dominance* property:

> **Property 1 (*Instruction Dominance*)**: any memory-update operation o executed by an event e in forward mode which updates a memory location a is said to *dominate* any other memory-update operation o' executed later by the same event e if and only if o' updates the very same memory location a.

Intuitively, we can exploit this property noting that if o and o' both update a, then in ub_e we will find first the reverse operation $\bar{o}'$ and then $\bar{o}$. Therefore, if o causes on memory location a the transition $a_1 \rightarrow a_2$, and o' causes the transition $a_2 \rightarrow a_3$, executing ub_e causes first the inverse transition $a_3 \rightarrow a_2$ and then $a_2 \rightarrow a_1$. Nevertheless, a discrete event can be seen as an *atomic unit of computation*. Therefore, if the event e is executed, either the value of a is updated from a_1 to a_3, or it needs to figure as not updated at all in case of rollback of e. Thereby, we can prevent ub_e to produce two inverse transitions: only the transition $a_3 \rightarrow a_1$ must be present within ub_e. Overall,

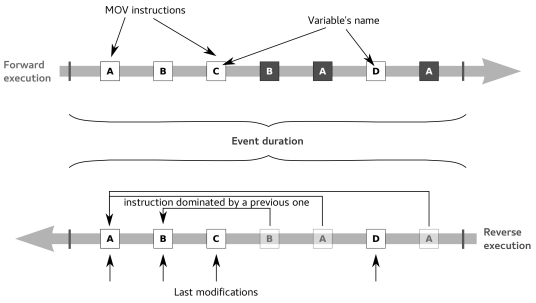


Fig. 2. Instruction dominance.

only the first memory update to a given location will lead to the generation of a reverse instruction. Any other memory update can be simply discarded, as shown in Figure 2.

2.2. Exploiting Locality: Per-chunk Undo Code Blocks

A further optimization to reduce both the overhead in forward execution mode (due to the generation of reverse instructions) and the overhead when executing an undo code block, comes directly from the way simulation models, and more in general software applications, are often implemented. Specifically, the states of the LPs may rely on structures or objects (that could point to other structures/objects), which are commonly stored in the heap, e.g., via calls to the `malloc` standard library or via the `new` operator.

In this case, we can extend **Property 1**, and define the *Object Dominance* property, where we use the term *object* in a broader way than the common usage in object-oriented programming, identifying any memory area that keeps a portion of the LP state and that was instantiated by a single memory allocation in the heap:

> **Property 2 (*Object Dominance*)**: any memory-update operation o executed by an event e in forward mode which updates a memory location a is said to *dominate* any other memory-update operation o' executed later by the same event e if and only if o' updates a memory location a' such that a and a' belong to the same object.

Intuitively, **Property 2** extends **Property 1** on the basis of the consideration that if a field of a structure, or an attribute of an object, is updated during the execution of an event, then other elements within the same (logically coupled) memory area will be likely updated as well. This is a consideration clearly related to classical locality principles for software applications.

For execution patterns where **Property 2** appears to be satisfied for a significant percentage of memory update operations, it could be more efficient to generate simpler undo code blocks. These can rely on compact instructions to restore the image of the whole structure/object as it was seen by the LP at the beginning of the execution of the event. We refer to this technique as *reversibility with per-chunk granularity*, a target that can be achieved creating a compact snapshot of the structure/object before the first memory-update operation falling within its address boundary is executed. This snapshot can be stored directly within the undo code block, along with instructions that copy it back on the original memory address[3]. The potential advantages of per-chunk reversibility arise especially in contexts where memory update operations dominated according to **Property 2** involve a non-minimal percentage of the chunk area, which

[3]We note that ISAs such as the x86 offer the optimized family of *string instructions* which can perform the restore operation of memory blocks very quickly.

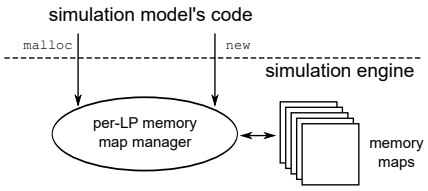


Fig. 3. Simulation engine organization with a per-LP memory-map manager

can be expected especially for small to medium chunks. We will come back to aspects related to the fragmentation of the operations within chunks in Section 3.4, where we discuss a heuristic for determining whether it is convenient to switch to per-chunk undo code blocks or not.

To make the detection of structures/objects involved in a memory-update operation completely transparent, the described static binary instrumentation technique alone is not sufficient. In fact, a memory-update operation can fall within a structure/object in any place, and determining the base address of the structure/object along with its size in memory is an aside problem with respect to the determination of the target location for the memory update. This problem can be tackled by introducing an additional layer to intercept all the calls to `malloc` functions or to the `new` operator, as shown in Figure 3. This approach has already been exploited in the context of checkpointing in Time Warp platforms to deliver memory to the application layer in a controlled manner, so that the exact memory map of LPs' states is known, and snapshots can be correctly taken when needed. In more detail, when the simulation model requests for memory, the underlying PDES engine can serve the request from buffers that are under the control of so-called *memory map managers*—many techniques exist to this end; see, e.g., [Bonwick 1994; Lea 1996; Peterson and Norman 1977; Toccaceli and Quaglia 2008]. These managers map any memory request to a suitably-sized *chunk of memory*.

These memory maps kept at the level of the PDES engine, along with instrumentation, allow us to put in place the per-chunk reversibility approach. Whenever a memory access falling within a structure/object composing the simulation state is detected, this access can be associated with one of the memory maps managed by the simulation engine, more specifically with a specific chunk (and its size). We propose two techniques for this association, both compatible with the generic map organization in Figure 3:

(1) *per-size memory maps, with aligned addresses*: the fastest possibility is to maintain multiple per-LP memory maps. Each map keeps same-sized chunks within the same memory segment. If chunk sizes are powers of two (which is typical), and the map is properly aligned in memory, then a simple bit-masking operation on the intercepted address will provide the base address of a chunk. To determine the number of bits to be masked, the size of the involved chunk is required. Nevertheless, the memory map manager knows the boundaries of the memory maps, and can infer the size by looking at the intercepted address. With this approach, care should be taken in allocating properly-sized maps to avoid map-internal (chunk external) fragmentation. Classical schemes based on dynamically doubling the size of a map associated with highly requested chunk sizes are generally adequate.
(2) *per-chunk hash tables*: a completely different solution is to use a hash table where a tuple $\langle base\ address, end\ address, size \rangle$ is stored. This hash table can be organized using the n most significant bits of $base\ address$ as the key. To obtain the base address and the size of a chunk from any address falling within it, the address targeted by the intercepted memory write instruction is masked so as to obtain the n most significant bits. This part of the address is then used to query the hash

table. To make a query conflict-free, any conflict-resolution technique (e.g., separate chaining) proper of hash tables could be exploited. Clearly, the hash table may significantly grow, depending on the number of chunks which are allocated. Moreover, in case of a `free`/`destroy` operation, the corresponding entries in the hash table should be removed/invalidated, leading to additional management cost. However, unlike the above approach, this one does not require pre-allocation of memory segments hosting the chunks, thus avoiding fragmentation.

When the simulation engine determines the base address of the updated chunk, an undo instruction is generated. The information associated with this generation can be maintained to avoid generating undo instructions for the same chunk by a same event.

In scenarios with good update locality, generating per-chunk undo code blocks in combination with the exploitation of dominance likely allows for reducing both the cost of the operations carried out along the critical path during the forward execution phase, and the cost to undo events during the reverse execution phase. We shall discuss in Section 3.4 how it is possible to devise an efficient implementation of the checks required by the per-chunk reversibility optimization, also exploiting a combination of the two different approaches in points (1) and (2).

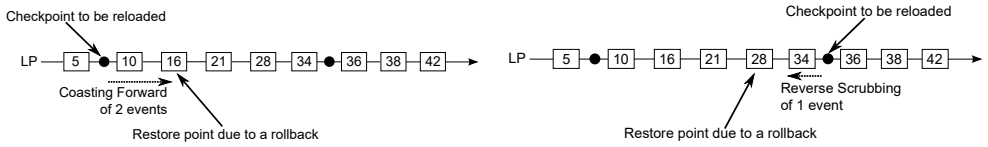
2.3. State Recovery with (not only) Undo Code Blocks: Concepts and Cost Analysis

A baseline scheme to exploit undo code blocks for state recovery upon the detection of a causality violation is to execute the undo code blocks associated with all the events to be rolled back, starting from the current live image of the LP state. However, this approach would make the recovery latency directly proportional to the length of rollback, which is unfavorable to performance in executions with long rollbacks. In order to make the state restore latency independent of the rollback length, the approach that needs to be exploited is to rely on infrequent checkpointing in combination with the undo code block approach.

Let us initially recall how pure (infrequent) checkpoint based approaches (see, e.g., [Preiss et al. 1994]) achieve restore latency that is independent of the rollback length. The state recovery procedure is based on identifying: (a) the last causally consistent event $\hat{e}$, associated with timestamp $T_{\hat{e}}$, and (b) a simulation state snapshot $\hat{S}$ associated with timestamp $T_{\hat{S}} \leq T_{\hat{e}}$. Clearly, after checkpoint $\hat{S}$ was taken, any number of events could have been executed before the execution of $\hat{e}$—this is the classical case when $T_{\hat{S}} < T_{\hat{e}}$. Hence, reloading $\hat{S}$ is not enough to bring the simulation state back to the restore time $T_{\hat{e}}$, as the effect of one or multiple events in the interval $(T_{\hat{S}}, T_{\hat{e}}]$ would have been undone as well. To realign the simulation state exactly to $T_{\hat{e}}$, the so-called *coasting forward* phase must be executed, which is a silent replay of the intermediate events in the interval $(T_{\hat{S}}, T_{\hat{e}}]$. An illustration of the recovery operation to simulation time $T_{\hat{e}}$ is reported in Figure 4(a).

As it is well known [Preiss et al. 1994], the cost associated with the coasting forward phase depends on two factors: the average granularity δ_e of any event to be replayed, and the actual number of the replayed events. Since when relying on periodic checkpointing a rollback operation can occur at any place in between two consecutive checkpointed states S_1 and S_2, denoting with χ the number of events which are executed between S_1 and S_2, the coasting forward phase requires to reprocess $\frac{\chi-1}{2}$ events on the average, thus leading to an average restore cost of $\delta_e \cdot \frac{\chi-1}{2}$. This makes state restore independent of the rollback length, and only dependent on the distance between checkpoints, and the average event granularity.

A similar combination of checkpoint reload and successive realignment can be employed when relying on undo code blocks, again in order to make the state recovery



(a) State recovery based on coasting forward (b) State recovery based on reverse scrubbing

Fig. 4. Different ways to support the state recovery operation

operation independent of the rollback length[4]. In such a case, rather than the coasting forward phase, we have to undertake what we call the *reverse scrubbing phase*. Particularly, we can reload the checkpoint $\hat{S}$ associated with a timestamp $T_{\hat{S}} \geq T_{\hat{e}}$, starting from which undo code blocks are then used to annihilate the effects of events[5]. This scenario is depicted in Figure 4(b), where the restore point (namely, event $\hat{e}$) is two events before the selected checkpoint $\hat{S}$. In this case, the rollback algorithm has to reload checkpoint $\hat{S}$, and then execute the undo code block of just one event. The cost of the reverse scrubbing phase depends on the average cost δ_{ub} for executing an undo code block, and the average number of events that have to be reverted, still $\frac{\chi-1}{2}$ when taking checkpoints each χ events. The average scrubbing phase cost is then $\delta_{ub} \cdot \frac{\chi-1}{2}$.

Let us now consider the relation between δ_e and δ_{ub}. Given that undo code blocks are not real reverse event handlers, since they revert side effects with no explicit recomputation of the values to be restored, we may expect that for complex event processing logics, δ_e would be greater than δ_{ub}. The consequence is that we may achieve, on average, the same state recovery cost via reverse scrubbing, compared to coasting forward, by taking periodic checkpoints less frequently.

However, relying on undo code blocks produces a forward computation overhead which is not present in classical approaches based only on periodic checkpointing. This is the overhead to generate at runtime undo code blocks, which makes the average event granularity to be no longer δ_e, rather $\delta_e + \delta_{inst}$, where δ_{inst} is the average per-event cost for executing the instrumentation code to generate undo code blocks.

In order to take the best of the two methodologies, including the benefits from the relation $\delta_e > \delta_{ub}$, a checkpoint interval χ can be selected where the initial portion $\chi - \nu$ of events in the interval are not covered by undo code blocks, while the remaining ν are. Then, either coasting forward is executed, if one of the uncovered events is to be restored, or reverse scrubbing, in the opposite case. Clearly, this approach is really effective only if no instrumentation cost is paid while processing the first $\chi - \nu$ events in any checkpoint interval. As we will discuss in Section 3, we achieve this via an optimized *multi-coding* scheme, still transparent to the application level software.

The combination of forward and backward restore capabilities within a checkpoint interval is a topic that has already been investigated, particularly in [Cortellessa and Quaglia 2001], however limited to the usage of incremental checkpoints (not undo code blocks) for backward recovery starting from the upper checkpoint within the interval. Re-adapting the performance model presented in that work to our context, we can obtain an expression to determine the expected overhead of both recoverability support, and of actual recovery operations. This expression is a function of the parameters discussed before, of the parameters δ_s and δ_r expressing the average cost for

[4]Full independence would need avoiding full backward traversal of checkpoint chains, which is an issue aside of the main techniques we are presenting.

[5]A corner case occurs when that checkpoint does not exist. In this situation we can apply the undo code blocks starting from the live state image of the LP.

taking/reloading a checkpoint, and of the LP rollback frequency F_r. Overall, the final re-adapted expression of the per-LP expected overhead OH is:

$$OH = \frac{(\delta_s + \nu \cdot \delta_{inst})}{\chi} + F_r \left[\frac{\chi - \nu}{\chi} \left(\delta_r + \frac{\chi - \nu - 1}{2} \delta_e \right) + \frac{\nu}{\chi} \left(\delta_r + \frac{\nu}{2} \delta_{ub} \right) \right] \quad (1)$$

In Equation (1), the independent parameters are χ and ν, since all the other parameters depend on the specific runtime dynamics of model execution. The values of χ and ν that minimize the overhead OH can be computed according to the method provided in [Cortellessa and Quaglia 2001]. However, one aspect that is uncovered by that method is whether the values δ_{inst} and δ_{ub} need to refer to the per-chunk reversibility approach or not. In fact, selecting one or the other approach is an issue intrinsically related to our undo code block based technique. As we will discuss in Section 3.4, our support for state recoverability entails a heuristic allowing the runtime selection of either per-chunk or baseline reversibility. Once the best-suited reversibility mode is selected, the runtime minimization of Equation (1) via the selection of χ and ν takes place by filling it with values of δ_{inst} and δ_{ub} sampled with the selected mode.

3. IMPLEMENTATION OF THE OVERALL STATE RECOVERABILITY ARCHITECTURE

As mentioned, we target Linux/x86_64 platforms and the ELF executable format. To instrument the application-level code, we rely on the Hijacker open-source static instrumentation tool [Pellegrini 2013]. For the purpose of this article, we have augmented the set of operations which this tool can perform.

Hijacker is conceived to be part of the compiling tool-chain, placing itself at a pre-linking stage. Our state recoverability architecture can be thus easily integrated into different simulation engines. The core support provided by Hijacker is to let the user specify (by using simple xml-based rules) how to manipulate the original application code. It allows to instrument at different scopes, namely executable-wide or at the level of single functions/instructions. Additionally, Hijacker allows creating multiple differently-instrumented copies of the same executable modules, packed within the same final executable. This technique, known as *multi-coding*, creates different versions of the code which share the same data sections within the virtual address space. Hijacker transparently allows changing the name of all the instrumented functions when it generates multiple versions of the software, by simply appending a user-defined suffix to them. This allows the simulation engine to identify which version of each function is instrumented in a specific way, or is not instrumented at all.

Overall, after the generation of multiple versions of the code, the final memory organization of the executable complies with Figure 5, where we provide an example with only two versions. This scenario is suited to support the recoverability scheme presented in Section 2.3, which is based on the combination of periodic checkpointing and undo code blocks. In fact, a non-instrumented version can be exploited to execute events that do not need reversibility, while an instrumented one can be used to run those events to be undone via reversibility.

3.1. The Instrumentation Scheme

To trace memory updates at runtime, we have specified the rules reported in Figure 6. This configuration instructs Hijacker to generate a modified application-level relocatable object which has two different versions (sharing the same data), one of which is by default the original non-instrumented code version. Specifically, by using the rules in the `<Executable>` tag, which generates the version associated with the `reverse` suffix, Hijacker scans the whole application 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 x86 instructions are `mov`,

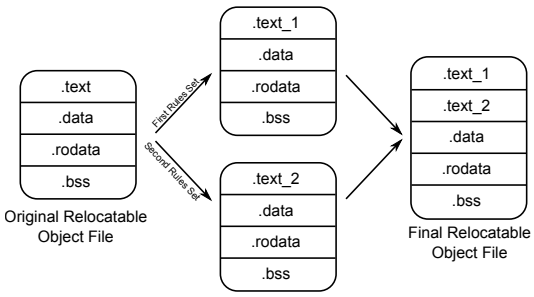


Fig. 5. Multi-coding scheme

```
<hijacker:Rules xmlns:hijacker="http://www.dis.uniroma1.it/~hpdcs/">
  <hijacker:Executable suffix="reverse"> <!-- instrumented code version -->
    <hijacker:Instruction type="I_MEMWR" skip="I_STACK">
      <hijacker:AddCall where="before" function="reverse_generator" arguments="target_address"/>
    </hijacker:Instruction>
  </hijacker:Executable>
</hijacker:Rules>
```

Fig. 6. Hijacker rules to instrument the application-level code

movs, and cmov instructions. 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 memory update instruction that has been instrumented. This is done via the <AddCall> tag, with the arguments attribute set to target_address. Memory-write instructions targeting locations into the stack (e.g., accessing memory using displacement from the stack pointer) are not instrumented in our rules. In fact, while processing whichever event via the associated handler, the stack is used as an area for transient data, that do not belong to the actual LP state, thus not needing recoverability of their updates.

According to the above instrumentation rule, Hijacker generates a cache of disassembly information, which can be exploited at runtime, particularly to avoid costly on-the-fly disassembly steps. In more detail, Hijacker extracts from the memory-write instruction the information related to the *size* of the memory area being updated, and the *destination address*. According to the x86 addressing mode, each memory address is identified by the expression $base\ address + (index \cdot scale) + displacement$, where 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. To cope with runtime evaluation of the address, while jointly avoiding on-the-fly disassembly, the compile-time retrieved data defining the address are used as a template that is then instantiated at runtime by simply pushing[6] the below described data structure, named insn_entry, onto the stack (see Figure 7):

```
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 compute the target address, or to identify the class of data-movement instructions; base keeps the (3 or

[6]In the appendix we illustrate the injected instructions that set up this information on the stack.

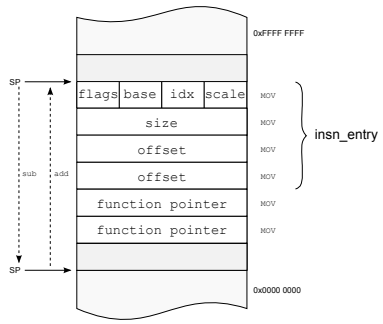


Fig. 7. Trampoline call stack frame

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 update instruction (when available at disassemble time); `offset` keeps the displacement of the addressing mode[7].

Additionally, Hijacker places onto the stack the address of the function specified in the `function` attribute of the `<AddCall>` tag. In our case, this is `reverse_generator(void *address, size_t size)`, which is in charge of generating the negative instruction that will undo the effect of the memory update on the LP state. Since this function was not present in the original executable[8], Hijacker simply creates a *relocation entry* in the final relocatable object file, leaving to the linker the task of identifying its correct address. At this point, the trampoline block of code, which is a part of the code injected via instrumentation, computes the actual value of `address`, and passes it as input to the `reverse_generator` function.

A few additional steps are carried out if the instrumented memory update was either a `cmov` (conditional-move) or a `movs` (move-string) instruction. Particularly, the `cmov` instruction is managed directly by the trampoline, which uses 4 bits in the `flags` field of `insn_info` to record what is the check to be emulated in order to determine if the conditional 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 snapshot of the status bits at trampoline startup are checked to determine whether the condition is met or not. If the check is positive, then control is passed to the `reverse_generator` function, exactly as for the case of classical `mov` instructions. In the negative case, 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, the `size` flag only tells the size of one single data move iteration (of the ones to be carried out for string-move). To compute the total size, the trampoline checks the value of the `rcx` register, and multiplies it by `size`. The starting address of the memory update is then computed by first checking the *direction flag* of the `flags` register. If this flag is zero, the destination address is already present in the `rdi` register. Otherwise, the `movs` will make a backwards copy, therefore the (logical) initial address of the move is computed as `rdi - rcx * size`. Then, the invocation of `reverse_generator` takes place.

[7]We provide 64-bit 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.
[8]In fact it is part of the state recoverability manager, not of the simulation model.

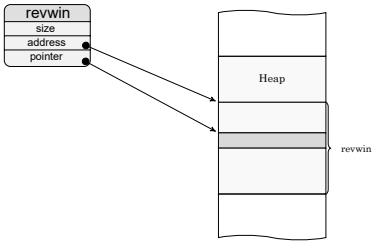


Fig. 8. Revwin descriptor

We emphasize that the trampoline has been developed directly in x86 assembly, in a very efficient way, thus reducing as much as possible the overhead to compute the target address. Additionally, the use of cached disassembly information allows the trampoline to access all the relevant information very quickly, which avoids the need for costly runtime disassembly.

Finally, as mentioned, the adopted 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 functions' versions by a name suffix, the underlying PDES engine can easily pass control to either instrumented or non-intrumented event handlers just depending on the target structure of the checkpoint interval of the LP.

3.2. Runtime Generation of Undo Code Blocks

The instrumentation architecture described so far allows to activate at runtime 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 intent to update the LP state, and therefore negative instructions can be built on-the-fly.

If the invocation of `reverse_generator` is related to the execution of a `mov` or a `cmov` instruction, the negative instruction is built by accessing memory at `address` and by reading `size` bytes. The "old" value is placed within a reverse data movement instruction (a `mov` instruction) as the source immediate operand. The destination of this negative instruction is `address`. On the other hand, if the activation of `reverse_generation` is due to a `movs` instruction, then the reverse instruction we put in the undo code block is another `movs`. We also note that the `movs` instruction, which allows for memory moving blocks of bytes, is what we have exploited for implementing the undo code block that restores a whole chunk when working with per-chunk reversibility.

Since the set of instructions to be generated within undo code blocks is very limited (namely, some variants of `mov` instructions), the opcodes are known beforehand. This allows us to use pre-compiled tables of instructions to be filled with relevant parameters, namely the old value and the destination address.

Our state recoverability architecture also offers the `initialize_event(int LP)` API that, when invoked by the PDES engine, informs the recoverability support that a simulation event is about to be delivered to some LP for processing. In this scenario, a fast cached allocation (backed from the heap), similar in spirit to the ones supported by the Linux slab allocator, is used to reserve room for storing the dynamically generated instructions. In more detail, these instructions are packed into a *reverse window* structure, which is depicted in Figure 8. When the reverse window is created, the reverse generator module places at the end of the associated memory area a `ret` instruction allowing the undo code block to return control to the caller function after

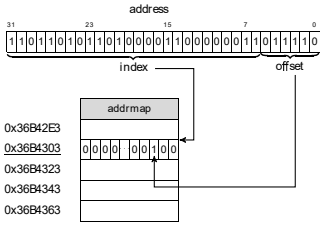


Fig. 9. The `index` and `offset` bitmasks of revwin's hashmap

its processing. Additionally, the `pointer` field of the `revwin` structure is initially set to the address of this `ret` instruction. Each time the `reverse_generator` module is called, the new memory-update reversing instruction is inserted right before the address of `pointer`, whose value is then updated accordingly. Therefore, the reversing instructions are placed into a stack forming the undo code block, whose activation takes place via a single `call` to `pointer` as the target[9].

3.3. Tracking Instruction Dominance

As mentioned, dominance properties are important for reducing both the generation cost of reversing instructions and the actual number of reverse instructions to be processed in a rollback phase. To determine whether memory write operations match *Instruction Dominance*, we employ an ad-hoc data structure to keep track of already referenced addresses (while processing an individual event), namely a fast hashmap.

Whenever `reverse_generator` is activated, this hashmap is queried to determine whether the destination address was already involved in a reversing instruction generation. Basically, this hashmap exploits a two-level bitmap to coalesce multiple addresses within a single word, so as to optimize space requirements 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 bi-dimensional 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[10]. 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 quadword. These are computed by properly masking the address value. A family range is therefore composed of all the addresses whose value starts with the same prefix. 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 9 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, the hashmap's content is flushed, so as to allow the new reverse window to store all the needed reversing instructions, even though previously-executed events touched the same memory addresses. Also, false positives are avoided by flushing a cache line each time a new family of addresses requires updating the line itself.

3.4. Tracking Object Dominance and Heuristic Switch to Per-chunk Undo Code Blocks

The above described hashmap is used in our implementation also for keeping track of *Object Dominance*, which is done by associating one hashmap entry to each chunk.

[9]The appendix discusses how to cope with memory exhaustion within the reverse window buffer.
[10]This allows us to handle both 64-bit x86 architectures and 32-bit ones, at the cost of wasting some space when running on older CPUs.

This is where we combine the two approaches presented in Section 2.2 in order to determine (upon any new memory update) whether a specific chunk has already been dirtied. In fact, each hashmap element is organized as a map of elements (already referenced or not) within a same chunk.

By keeping track of *Object Dominance* this way, we can jointly put in place a simple heuristic for determining whether to switch the operating mode of the undo code block-based recoverability support to per-chunk granularity, or not. Specifically, the information kept by the hashmap entry associated with a chuck can be used to represent the so called *chunk fragmentation*. This is evaluated as the percentage of elements of the chunk, which have not been accessed in write mode. If the fragmentation factor fr_i for the i-th chunk falls below a given threshold α (set in the interval $[0, 1]$) then locality of the accesses in write mode to different portions of that chunk is to be considered low. Therefore, it would not be convenient to rely on per-chunk undo code blocks. Clearly, the choice on whether to switch to per-chunk undo code blocks needs to consider an average fragmentation factor (across all the chunks) that can be easily computed as:

$$FR = \frac{\sum_1^n fr_i}{n} \tag{2}$$

As a general rule, for applications mostly using small-size chunks, higher values of α could lead to better performance tradeoffs between the cost of per-chunk reversibility and that of basic reversibility of memory updates on individual locations.

4. EXPERIMENTAL ASSESSMENT

4.1. Test-bed Platform

Our undo code block-based reversibility support has been fully integrated within the open source ROOT-Sim package [Pellegrini and Quaglia 2013], which we use as the reference PDES environment in this experimental study. ROOT-Sim already supports fully-transparent checkpointing (see [Pellegrini et al. 2015]), based on the DyMeLoR memory map manager [Toccaceli and Quaglia 2008]. This same memory map manager has been exploited while integrating our undo code block approach to generate the combined (checkpoint/reversibility-based) restore technique presented in Section 2.3. Our parallel runs have been carried out on a 32-core HP ProLiant server (equipped with 64GB of RAM) running Debian 6 on top of the 3.16.7 Linux kernel. A ROOT-Sim configuration with 32 worker threads has been used in all the experiments, with Global-Virtual-Time (GVT) computation and fossil collection (of committed data records and of no longer useful recoverability data) taking place every second.

4.2. The Test-bed Application

As test-bed application, we have selected a cellular system simulation model. The significance of this model is due to its complex simulation workload, which is also highly variable (e.g., in terms of memory access pattern to the states of the LPs, and intensity of the write activity on these states) depending on how it is configured. Overall, this test-bed application allows us to assess our recoverability support by exploiting a suite of different workload profiles.

In this application, each cell is modeled as a hexagonal coverage-area whose evolution is simulated by an individual LP, by simulating the evolution of the state of its hosted N radio channels. Each channel is modeled in a high fidelity fashion by relying on the results provided in [Kandukuri and Boyd 2002], which state how interference and fading phenomena affect the experienced Signal-to-Interference Ratio (SIR). Accordingly, power regulation is actuated in order to achieve predetermined SIR levels, which is also explicitly modeled.

The current state of any channel is kept via a dynamically-allocated record, placed into a list. Also, two variants for the LP state layout have been considered, exhibiting different tradeoffs. In the first variant, each channel record points to a power-management record, which is also dynamically allocated upon call setup. In the second variant, power-management records are dynamically allocated in block, and are used to serve multiple call installations. Particularly, a block is made up by 100 records, and the different stocks are recorded within a hash-table, which is indexed by relying on the channel-identifier. We will refer to the second variant as *Clustered*. This variant allows for improved memory locality (since multiple power-management records belong to the same contiguous chunk), at the expense of internal memory fragmentation (within the LP state) if a reduced percentage of entries are used within a block.

The latter aspect is linked to the workload intensity of simulated calls, which is another parameter we have varied in our experiments. In more detail, the model includes a parameter τ_A, which expresses the inter-arrival time of subsequent channel assignment requests to any target cell. The lower τ_A, the higher the workload. An additional parameter $\tau_{duration}$ expresses the expected duration of a communication burst on a channel after its assignment; hence the channel *utilization factor* can be computed as $\tau_{duration}/(\tau_A \cdot N)$. Higher utilization factors require handling more power-management records at any time instant. This impacts the event granularity (which depends on the number of channel records and power-management records to be scanned and/or updated) and the memory footprint of the LP state. The additional parameter τ_{cell_switch} is used to express the residual residence time of a mobile device into the current cell. This parameter determines, in combination with $\tau_{duration}$, the interactions across the LPs, in the form of hand-off events cross-scheduled among them.

In our study, we consider a scenario with 1024 cells covering a square region, each one managing $N = 1000$ wireless channels (resembling macro-cell technology). $\tau_{duration}$ is exponentially distributed with average value 120 seconds, while τ_{cell_switch} is exponentially distributed with average value 300 seconds. Also, the inter-arrival time τ_A (still exponentially distributed) is varied in terms of its mean value in order to achieve two different values for the average wireless channels' utilization factor, namely 25% and 75%. This diversifies the actual execution pattern (CPU/memory demand) by the application. These two scenarios will be referred to as *Low-Load* and *High-Load*.

Further, we consider two different read/write operations profiles. The first one is based on computing the SIR value only when performing power regulation upon new call setup. We will refer to this profile as *Read-Intensive*, since channel/power-management data records are scanned (but not updated) while determining power regulation for the new call. A second profile is instead based on updating SIR values (only depending on fading) for all the on-going calls upon new call setup. This profile leads to updating all the active power-management records upon the installation of any call, hence we will refer to it as *Read-Write* profile.

On the basis of the varied parameters, a total number of six different configurations have been run, which are summarized in Table 4.2. The Clustered variant of the LP state layout has been tested only in the Read-Write profile, given that it is intrinsically tailored to capturing the effects on the recoverability architecture by locality of the updates (if any) into the LP state layout.

As a last preliminary note, the average event execution times have been observed to be of the order of 80–100 μs for LL profiles, and of the order of 150–200 μs for HL profiles. The time to take a checkpoint has been observed to be of the order of 30 μs for LL profiles and 55 μs for HL profiles (slightly smaller values have been observed for the Clustered setting). The average time to generate a single reversing instruction for an individual 64-bit memory location has been observed to be of the order of 0.15 μs.

Table I. Summary of test-bed configurations

	Read-Intensive	Read-Write
Low-Load	RI-LL	RW-LL
High-Load	RI-HL	RW-HL
Low-Load Clustered	—	C-LL
High-Load Clustered	—	C-HL

4.3. Performance Data

In a first set of experiments, we compare the performance of either traditional periodic checkpointing, in combination with coasting forward for restoring the LP state to non-logged values, or of periodic checkpointing in combination with reverse scrubbing. As explained in Section 2.3, the latter recovery scheme is based on either applying the undo code blocks starting from the current state image of the LP, or by applying them after reloading a conveniently-selected checkpoint. Also, for the configuration based on reverse scrubbing, we generate undo code blocks for all the processed events, thereby excluding the possibility of combined usage of coasting forward and reverse scrubbing within a same checkpoint interval. The effects of this combination, and the optimization of the combination on the basis of the cost/benefit model in Equation (1), will be assessed later via a second set of experiments.

For completeness, we also consider a configuration where state recoverability is based on pure reverse scrubbing (no checkpoint is taken). Further, for all the configurations entailing the generation of undo code blocks, we run the experiments by either activating or disabling the hashmap to track already-updated memory locations (or memory objects) in the processing of a single event. This allows us to evaluate overhead and benefits by the hashmap (via the exploitation of dominance). The hashmap is configured with 128 lines. Data referring to the pure reverse scrubbing configurations are shown as bars, since they are independent of the checkpoint interval.

We report in Figure 10 the execution speed variation of the simulation (evaluated as the amount of committed simulation time units per second) for both the RI-LL and RI-HL configurations[11]. The curves for reverse scrubbing, either in combination with checkpointing or not, refer to the case of baseline undo code block generation (see Section 2.1), where every memory update is reverted via a corresponding memory-move instruction. For the RI-LL configuration, we observe similar peak performance for both coasting forward and reverse scrubbing. However, reverse scrubbing shows higher resilience to performance degradation vs. sub-optimal values of the checkpoint interval. This is somehow expected, given that the RI profile produces undo code blocks with very few instructions. Thus, the reconstruction of non-checkpointed state images is a reduced-cost operation even if the distance between the state image to be recovered and the starting point of the reverse scrubbing operation is non-minimal. This is also evident when looking at the performance provided by pure reverse scrubbing, which stands slightly better than the configuration with reverse scrubbing in combination with checkpointing when the interval between checkpoints is set to 40. Good resilience to performance degradation is instead not achieved by classical periodic checkpointing with coasting forward. In fact, this configuration shows a rapid performance increase when moving the checkpoint period from one to five events, thanks to a significant reduction of the checkpointing overhead. However, for larger values of the checkpoint period, the additional reduction of the checkpointing overhead does not pay off, since the longer expected coasting forward degrades performance due to the need for replaying events to restore the target state, which is instead avoided by reverse scrubbing.

[11]All reported samples refer an average value computed over 5 different runs of a same configuration.

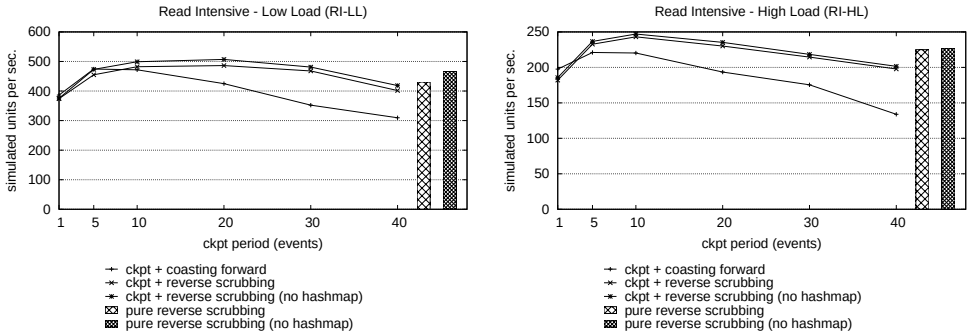


Fig. 10. Results for the Read-Intensive profile

Another interesting point is that, for checkpoint period set to one, both classical coasting forward and reverse scrubbing only need to reload one checkpointed state image upon a rollback operation. However, for the reverse scrubbing configuration all the processed events give rise to their associated undo code blocks. Hence, the reduced performance by reverse scrubbing vs. coasting forward in this configuration is representative of the extra cost for generating undo code blocks and tracing memory write operations. This cost does not allow reverse scrubbing to achieve the same performance boost as coasting forward when moving a checkpoint interval of five events—in fact the performance curve of reverse scrubbing is slightly smoother. However, the cost to generate undo code blocks pays off for larger values of the checkpoint interval, as discussed. Concerning the exclusion of the hashmap with reverse scrubbing, it leads to a slight performance increase compared to the scenario when it is used. This is because the RI configuration has no advantages from the hashmap, since one event updates memory locations belonging to the LP state only once. Hence, querying the hashmap leads to misses with high likelihood. On the other hand, this is a good test case to assess the cost for manipulating the hashmap, especially in relation to the operation of resetting it at the beginning of the processing phase of any event, given the low intensity of write operations within the LP state.

For the RI-HL configuration, the advantages by reverse scrubbing are amplified. In fact, HL gives rise to coarser-grain events which are adverse to coasting forward, especially for sub-optimal values of the checkpoint interval. In more detail, avoiding to take checkpoints at each event does not provide the same performance improvement as compared to RI-LL. Indeed, the costs for replaying coarser-grain events in coasting forward become non-negligible, as soon as the length of the coasting forward phase is non-minimal. As hinted, reverse scrubbing avoids at all these costs. Hence, thanks to the RI profile generating per-event undo code blocks with reduced numbers of instructions, the reverse scrubbing phase is still not costly. This allows for a better performance boost while increasing the checkpoint period. Overall, the reverse-scrubbing peak performance is about 10% higher than the coasting-forward peak performance. Further, similarly to the RI-LL configuration, reverse scrubbing still provides better resilience to performance degradation vs. sub-optimal values of the checkpoint period. Also, pure reverse scrubbing gives a performance that is slightly better than when using checkpointing with an interval set to 40. This is an interesting phenomenon caused by the fact that when the checkpoint interval is longer, reverse scrubbing is forced to reprocess more undo code blocks, and the latency to restore the last correct checkpoint form which to apply the undo code blocks does not pay off, compared to state reconstruction via pure reverse scrubbing. This is clearly linked to (and amplified by) the larger state footprint of the HL configuration, as compared to LL, which

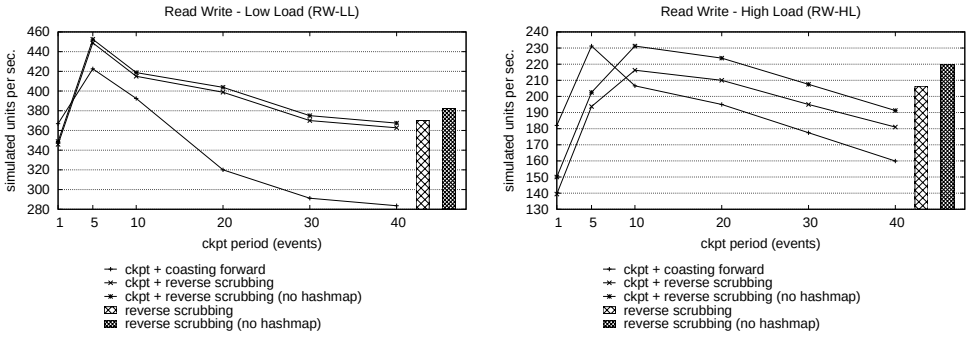


Fig. 11. Results for the Read-Write profile

leads to larger state/checkpint size, and more costly checkpoint reload. Finally, with more costly events, which lead to query the hashmap infrequently given the RI profile of memory interactions, the overhead for managing the hasmap tends to disappear, as compared to the LL configuration.

In Figure 11, we show the results from the RW profile. In this scenario, reverse scrubbing still undoes memory updates via single memory-move instructions placed into the undo code block. Compared to RI, this profile leads to increased frequency of per-event generation of undo code blocks, given that more memory-update operations are carried out by the events. For RW-LL, reverse scrubbing allows for better peak performance (with a gain of about 5% compared to coasting forward), and this time the performance of reverse scrubbing has a trend similar to coasting forward, with a peak speed at checkpoint period set to five. This is due to increased frequency of write operations, leading to larger per-event undo code blocks, whose generation/ processing costs penalize the state restore latency for longer reverse scrubbing phases (similarly to what happens with longer coasting forward phases). However, reverse scrubbing still looks more resilient to performance degradation for sub-optimal values of the checkpoint period (namely, excessively longer checkpoint periods). In fact, the pure reverse scrubbing configuration does not degrade performance, if compared to reverse scrubbing used in combination with checkpointing and large checkpoint intervals. For RW-HL, coasting forward is instead able to provide better peak performance (with a gain of about 6% over reverse scrubbing). This trend is due to RW-HL leading to further increased intensity of per-event write operations (compared to RW-LL), since a larger number of records are updated while processing the events. This makes memory-tracing and undo code block generation overheads the dominating factors. In fact, for checkpoint interval set to one, i.e. when the undo code block generation represents pure overhead with no benefit, we have a performance decrease of reverse scrubbing vs. coasting forward of about 33%. However, for longer checkpoint periods (beyond the optimal value for the coasting forward configuration), avoiding at all to replay events thanks to reverse scrubbing still pays off, leading to better performance compared to coasting forward. Also, the usage of pure reverse scrubbing again pays off compared to the combination of reverse scrubbing with checkpointing and large checkpoint intervals, because of the already-discussed reasons and possibly a decrease of locality caused by the need for restoring larger states while still needing long reverse scrubbing phases involving a higher number of machine instructions within undo code blocks touching memory sparsely. Finally, in the RW configuration we see a more pronounced difference in the performance achieved with the inclusion or exclusion of the hasmap. We remind that the RW profile still leads to memory updates mostly scattered on different locations, which prevents the hashmap from providing significant benefits.

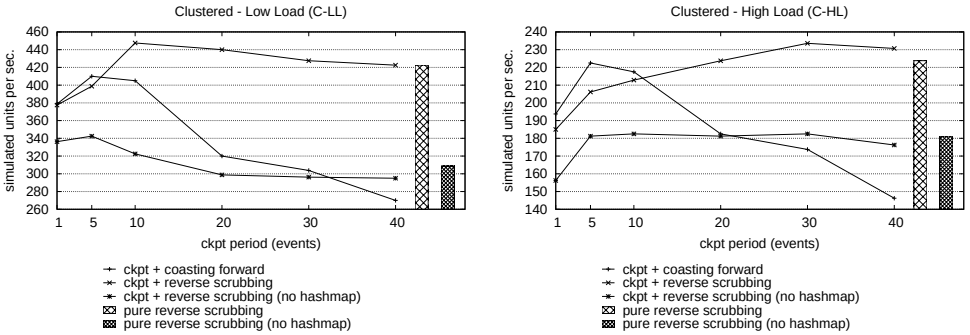


Fig. 12. Results for the Clustered configuration (RW profile)

On the other hand, the higher write intensity in the RW configuration (compared to RI) leads to querying the hashmap more frequently, which leads to an increase in the overhead. Such phenomenon has a higher relative incidence on HL due to the higher relative cost to manipulate and update records within the LP state vs. other operations carried out by the events.

As hinted, in the baseline memory layout of the benchmark application, the higher intensity of write operations in the RW profile is implicitly coupled with reduced locality of the updates, since the data records that are updated are scattered in memory. The Clustered variant of the memory layout is aimed at assessing the undo code block based recoverability support when employing per-chunk undo code blocks (recall that chunks represent this time blocks of records). We show in Figure 12 the execution speed curve for the Clustered configuration for both classical coasting forward and reverse scrubbing (either in combination with checkpointing or not) with undo code blocks based on per-chunk granularity. In the latter configuration, a single `movs` instruction restores the whole content of a block. Also, when the hashmap is employed, multiple updates on a same block performed while processing an event are discarded (in terms of generation of reverse instructions within the undo code block) thanks to the Object Dominance property. By the results, we see how the exploitation of locality of the memory updates, via the reliance on per-chunk undo code blocks, allows reverse scrubbing to provide performance improvements for the Clustered configuration (which exhibits a RW profile) in both LL and HL scenarios. Also, compared to the previous configurations, the resilience to performance degradation by reverse scrubbing is even more evident, given that longer checkpoint intervals lead to reduced amounts of reverse instructions to be processed in a rollback phase, since a single reverse `movs` instruction allows to restore the state of multiple records within the LP state. Overall, when some locality is guaranteed for memory update operations, reverse scrubbing is competitive even with higher intensity of memory write operations by the events. On the other hand, the exclusion of the support for tracing such locality, i.e. the hashmap, leads reverse scrubbing to provide definitely reduced performance caused by the work for generating/processing undo code blocks. Overall, saving the cost to manage the hashmap does not pay off in scenarios with actual dominance. As a last note, the usage of pure reverse scrubbing in combination with the hashmap, although revealing highly efficient, does not outperform reverse scrubbing in combination with checkpointing with larger checkpoint intervals. This phenomenon is due to the fact that with clustered allocation a checkpoint is more compact, since a reduced amount of metadata are involved. Hence, the cost of restoring a checkpoint from which to apply undo code blocks still pais off thanks to the statistical incidence of the reduction of the length of the reverse scrubbing phase (compared to pure reverse scrubbing).

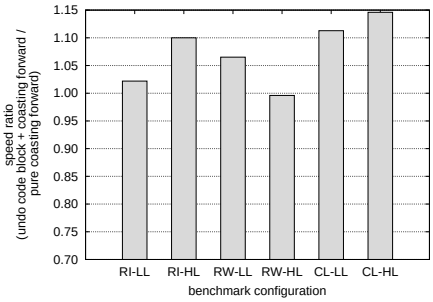


Fig. 13. Execution speed ratio (new approach vs traditional periodic checkpointing)

In the final part of this study we provide data related to the combined usage of reverse scrubbing and traditional coasting forward, according to the scheme proposed in Section 2.3. The potential for further performance improvements when generating checkpoint intervals where an initial portion of the events do not generate undo code blocks (i.e. restore is based on coasting forward) and a subsequent portion generate undo blocks (i.e. restore is based on reverse scrubbing) looks evident when considering that the undo code block technique makes the recovery latency less influenced by the distance between checkpoints. So we might set longer checkpoint intervals, while jointly avoiding long reverse scrubbing phases thanks to coasting forward limited to the initial, possibly reduced, portion of the checkpoint interval. To assess the advantages by the combined approach, we have run the same six configurations of the benchmark by optimizing the partitioning of the checkpoint interval according to the model in Equation 1. In our runs, the model is resolved at runtime after having collected data for its input parameters, which are also used to determine whether to switch to per-chunk based undo code blocks. This is done using Equation (2) after setting the fragmentation factor threshold α to the value 0.75 (we remind that it determines whether a chunk of memory has to be considered as fragmented or not, in terms of the updates occurring on it). The competitor configuration is identified as one based on periodic checkpointing, where the checkpoint period is still selected at runtime as the one minimizing Equation (1) once fixed the parameter ν to zero. With this settings, the equation boils down to one determining the optimal periodic checkpoint interval in traditional coasting forward based recoverability [Rönngren and Ayani 1994].

The achieved results are plotted in Figure 13. We report the ratio between the execution speed when using the mixed technique and traditional periodic checkpointing. The data refer to the steady state speed observed after the model-based performance optimization is already put in place. We simply discarded the initial speed samples collected when (for both recoverability techniques) no optimization of the parameters driving their behavior was already actuated. By the data, we see how the performance benefits from reverse scrubbing in combination with coasting forward are further amplified, especially for RI-HL, and for the C-LL and C-HL configurations (the latter being both RW in their profile). Further, relying on the model-based optimization allows the integrated recoverability support to avoid performance penalties when running applications with more intense memory write activity and with updates that are scattered (thus not favoring locality). In fact, for the RW-HL configuration, we observe that the model-based approach leads to excluding the usage of undo code blocks (thus leading to a pure checkpoint based recoverability scheme), hence avoiding the costs (and penalties) associated with the reverse scrubbing approach evidenced by the data in Figure 11.

5. RELATED WORK

Due to its fundamental role in optimistic PDES, state recoverability is a deeply investigated topic. As for checkpoint-based recoverability, 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) [Fleischmann and Wilsey 1995; Preiss et al. 1994; Quaglia and Santoro 2003; Rönngren and Ayani 1994], or incrementally logging modified state portions [Rönngren et al. 1996; Steinman 1993; West and Panesar 1996], or supporting a mix of the two approaches [Franks et al. 1997; Soliman and Elmaghraby 1998]. Most of these solutions demand from the application-model developer the implementation of state-saving 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 [Das et al. 1994; Steinman 1992; Toccaceli and Quaglia 2008] address the management of dynamic memory maps forming the LP state, and how to (transparently) log/restore them. Our proposal is complementary to all the aforementioned ones, as we integrate traditional checkpointing with the runtime generation of undo code blocks.

The general concept of using logs of instructions for reverse execution is not novel [Frank 1999; Biswas and Mall 1999; Sosič 1994], although we are not aware of works dealing with the speculative PDES context, where other reversibility schemes have been investigated. A recent work [LaPre et al. 2014] 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 [LaPre et al. 2014], 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 the events 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. The final tradeoff by our solution is different from the one in [LaPre et al. 2014] also because our instrumentation scheme leads to the runtime tracing of memory write operations, thus inducing some cost for the dynamic generation of undo code blocks. A reverse-computing based approach working on executables is provided in [Hou et al. 2012b], where the authors implement the reverse execution logic by automatically inspecting data dependencies within executable images. This approach is still a pure reverse-computing one, while our proposal reverts memory updates with no backward re-computation of the values to be restored.

The work in [Pellegrini et al. 2009], similarly to what we do, relies on static binary instrumentation to track memory updates during the forward execution of the events. Nevertheless, its goal 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 [West and Panesar 1996], 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.

In [Schordan et al. 2015] the address and old value of each updated memory location is saved during forward execution. Nevertheless, in this approach binary instructions are not generated at runtime. Consequently, the approach has to loop over saved data to restore the state during rollback, possibly adding a cost compared to our undo code

block based scheme. At the same time, the proposal in [Schordan et al. 2015] is based on Backstroke [Hou et al. 2012a], a C++ source-to-source translator based on the ROSE compiler, which can provide the advantage of enabling the instrumented code to rely on compiler-based optimizations.

As for the possibility to mix together different recoverability techniques (say checkpointing and undo code blocks in our case), our approach shares underlying principles with the works in [Cortellessa and Quaglia 2001; Pellegrini et al. 2015]. In [Pellegrini et al. 2015], 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 enabling the recoverability of a subset of the states passed through in the interval via update undo code blocks). Hence we target the optimization of runtime dynamics at smaller granularity levels (single events within a checkpoint interval). With respect to [Cortellessa and Quaglia 2001], we rely on a similar model for the optimization of the parameters driving the operation of the recoverability architecture. However, in that work no support for application transparency of recoverability tasks is presented, while our proposal is fully application transparent.

Our approach is also related to proposals in the field of program execution tracing (see, e.g., [gdb ; Bala et al. 2000; Qin et al. 2006; Zhao et al. 2008]). They provide detailed analysis of changes in the state and/or execution flow of programs. 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, like PDES. Debugging facilities exhibiting basic operating modes comparable to the one we employ (namely, the usage of trap mechanisms based on code injection and/or replacement to detect memory write accesses) are those addressing data watch points (see, e.g., [Wahbe et al. 1993]). 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 or, more in general, recoverability operations are not considered.

6. CONCLUSIONS

In this article we have presented a new technique for enabling state restore of simulation objects in speculative PDES systems. It is based on the runtime generation of undo code blocks, which are blocks of machine instructions able to revert the memory updates performed by incorrectly executed simulation events along the speculative execution path. The philosophy at the core of our proposal is to combine undo-logs with software reversibility concepts, since undo code blocks are compact logs of machine instructions (rather than of data). Our approach operates transparently, via automatic instrumentation of the application code, particularly ELF objects on x86/Linux systems. Also, the new recoverability support can be combined with classical checkpointing, which allows for exploiting the best of both worlds. This is done via the generation of a recoverability architecture that is able to restore a passed through state by either reloading a checkpoint, and fictitiously reprocessing intermediate events, or by applying undo code blocks starting from the current state image (or from a conveniently selected checkpoint). This improves the flexibility in determining the length of the checkpoint period, and the subset of events for which undo code blocks should be (or should not be) generated within any period, for performance optimization. We tested our approach on a suite of application execution profiles generated via parame-

terizing a mobile communication simulation application. The resulting data show how the undo code block approach has good potential for performance improvements with highly differentiated workloads.

REFERENCES

GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. *SIGPLAN Notices 35,* 5, 1–12.

BELLENOT, S. 1992. State skipping performance with the Time Warp operating system. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS)*. 53—-64.

BISWAS, B. AND MALL, R. 1999. Reverse execution of programs. *SIGPLAN Notices 34,* 4, 61–69.

BONWICK, J. 1994. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*. 6.

CAROTHERS, C. D., PERUMALLA, K. S., AND FUJIMOTO, R. M. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation 9,* 3, 224–253.

CORTELLESSA, V. AND QUAGLIA, F. 2001. A checkpointing-recovery scheme for time warp parallel simulation. *Parallel Computing 27,* 9, 1227–1252.

DAS, S. R., FUJIMOTO, R. M., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: a Time Warp system for shared memory multiprocessors. In *Proceedings of the 26th conference on Winter simulation*. Society for Computer Simulation International, 1332–1339.

DI SANZO, P., QUAGLIA, F., CICIANI, B., PELLEGRINI, A., DIDONA, D., ROMANO, P., PALMIERI, R., AND PELUSO, S. 2015. A Flexible Framework for Accurate Simulation of Cloud In-Memory Data Stores. *Simulation Modelling Practice and Theory*.

FLEISCHMANN, J. AND WILSEY, P. A. 1995. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 50–58.

FRANK, M. P. 1999. Reversibility for efficient computing. Ph.D. thesis, Massachusetts Institute of Technology.

FRANKS, S., GOMES, F., UNGER, B., AND CLEARY, J. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 72–79.

FUJIMOTO, R. M. 1990. Performance of Time Warp Under Synthetic Workloads. In *Proceedings of the Multiconf. on Distributed Simulation*. Society for Computer Simulation, 23–28.

FUJIMOTO, R. M., TSAI, J. J., AND GOPALAKRISHNAN, G. 1992. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers 41,* 1, 68–82.

HOU, C., VULOV, G., QUINLAN, D., JEFFERSON, D., FUJIMOTO, R., AND VUDUC, R. 2012a. *A New Method for Program Inversion*. Springer Berlin Heidelberg, Berlin, Heidelberg, 81–100.

HOU, C., VULOV, G., QUINLAN, D. J., JEFFERSON, D., FUJIMOTO, R., AND VUDUC, R. W. 2012b. A new method for program inversion. In *Proceeding of the 21st International Conference on Compiler Construction*. 81–100.

JEFFERSON, D. R. 1985. Virtual Time. *ACM Transactions on Programming Languages and System 7,* 3, 404–425.

KANDUKURI, S. AND BOYD, S. 2002. Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications. *IEEE Transactions on Wireless Communications 1,* 1, 46–55.

LAPRE, J. M., GONSIOROWSKI, E. J., AND CAROTHERS, C. D. 2014. LORAIN: A Step Closer to the PDES 'Holy Grail'. In *Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*. PADS. ACM Press, 3–14.

LEA, D. 1996. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html.

LIN, Y.-B. AND LAZOWSKA, E. D. 1990. *Reducing the saving overhead for Time Warp parallel simulation*. University of Washington Department of Computer Science and Engineering.

PALANISWAMY, A. C. AND WILSEY, P. A. 1993. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and distributed simulation*. ACM, 127–134.

PELLEGRINI, A. 2013. Hijacker: Efficient static software instrumentation with applications in high performance computing. In *Proceedings of the 2013 International Conference on High Performance Computing and Simulation, HPCS 2013*. Helsinki, Finland, 650–655.

PELLEGRINI, A. 2015. *Parallelization of Discrete Event Simulation Models*. Sapienza Università Editrice, Rome, Italy.

PELLEGRINI, A. AND QUAGLIA, F. 2013. The ROme OpTimistic Simulator: A Tutorial. In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*. PADABS. LNCS, Springer-Verlag.

PELLEGRINI, A., VITALI, R., AND QUAGLIA, F. 2009. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*. IEEE, 45–53.

PELLEGRINI, A., VITALI, R., QUAGLIA, F., PELLEGRINI, A., AND QUAGLIA, F. 2015. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems 26,* 6, 1560–1569.

PETERSON, J. L. AND NORMAN, T. A. 1977. Buddy systems. *Communications of the ACM 20,* 6, 421–431.

PREISS, B. R., LOUCKS, W. M., AND MACINTYRE, D. 1994. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Transactions on Modeling and Computer Simulation 4,* 3, 223–253.

QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 135–148.

QUAGLIA, F. 2001. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems 12,* 4, 346–362.

QUAGLIA, F. AND SANTORO, A. 2003. Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation. *IEEE Transactions on Parallel and Distributed Systems 14,* 6, 593–610.

ROBINSON, S. 2004. *Simulation: The Practice of Model Development and Use*. John Wiley & Sons.

RÖNNGREN, R. AND AYANI, R. 1994. Adaptive Checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*. Society for Computer Simulation, 110–117.

RÖNNGREN, R., LILJENSTAM, M., AYANI, R., AND MONTAGNAT, J. 1996. Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 70–77.

SCHORDAN, M., JEFFERSON, D., BARNES, P., OPPELSTRUP, T., AND QUINLAN, D. 2015. *Reverse Code Generation for Parallel Discrete Event Simulation*. Springer International Publishing, Cham, 95–110.

SEAL, S. K. AND PERUMALLA, K. S. 2011. Reversible parallel discrete event formulation of a tlm-based radio signal propagation model. *ACM Trans. Model. Comput. Simul. 22,* 1, 4.

SKOLD, S. AND RÖNNGREN, R. 1996. Event Sensitive State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference*. Society for Computer Simulation, 653–660.

SOLIMAN, H. M. AND ELMAGHRABY, A. S. 1998. An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation. *IEEE Transactions on Parallel and Distributed Systems 9,* 10, 947–951.

SOSIČ, R. 1994. History cache: Hardware support for reverse execution. *SIGARCH Computera Architecture News 22,* 5, 11–18.

STEINMAN, J. S. 1992. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation. *International Journal on Computer Simulation*, 251–286.

STEINMAN, J. S. 1993. Incremental State Saving in SPEEDES Using C Plus Plus. In *Proceedings of the Winter Simulation Conference*. Society for Computer Simulation, 687–696.

TOCCACELI, R. AND QUAGLIA, F. 2008. 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*. IEEE Computer Society, 163–172.

WAHBE, R., LUCCO, S., AND GRAHAM, S. L. 1993. Practical Data Breakpoints: Design and Implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1–12.

WEST, D. AND PANESAR, K. 1996. Automatic Incremental State Saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 78–85.

ZHAO, Q., RABBAH, R., AMARASINGHE, S., RUDOLPH, L., AND WONG, W. F. 2008. How to do a million watchpoints: Efficient Debugging using dynamic instrumentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4959 LNCS*, 147–162.

A. APPENDIX

A.1. Effect of the Instrumentation Process on the Program Image

In this section we discuss what is the nature and the amount of instructions injected by our instrumentation technique in order to activate the `reverse_generator()` module. This discussion and analysis allows the reader to fully understand how our technique guarantees the correct execution of an instrumented application, and to perceive what is the execution cost associated with this technique.

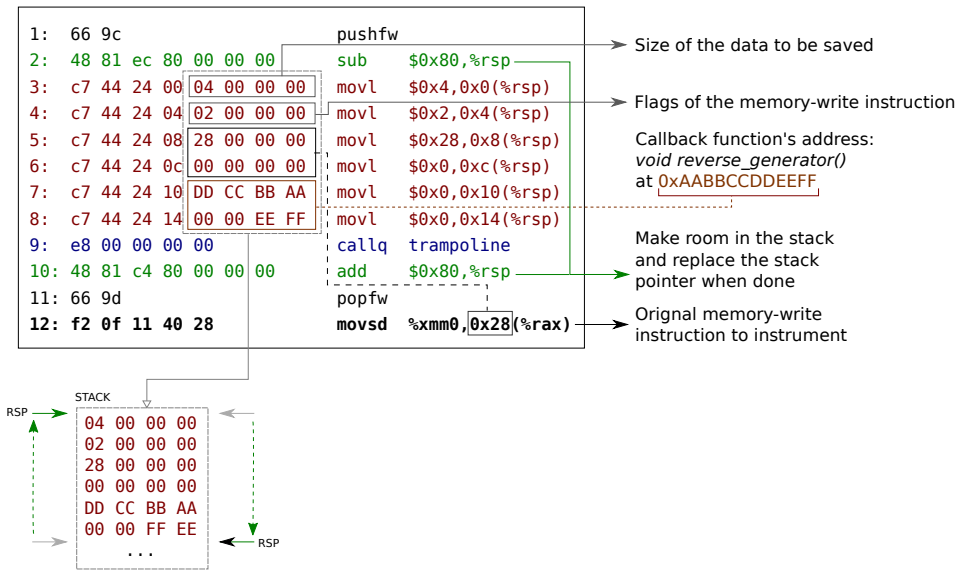


Fig. 14. Illustration of the instructions injected by the instrumentation process

Figure 14 illustrates an example of the outcome of the instrumentation process, along with the effect on stack by the execution of the injected instructions. The provided example refers to the `movsd %xmm0, 0x28(%rax)` x86 instruction, which is usually generated by compilers to store the result of a floating-point instructions into a field of a C `struct`. As mentioned, the instrumentation process installs a call to a `trampoline` function which activates a generic routine specified in the Hijacker's configuration file. In our case, this routine is `reverse_generator()`, which generates the reverse instruction for the `movsd` instruction, in this case.

To this end, the `trampoline` function requires the entry address of `reverse_generator()` to call it at runtime—in this example, this function is located at virtual address `0xAABBCCDDEEFF`. This information is placed on stack, for later retrieval by `trampoline`. Similarly, we place on stack additional information generated at compile time by Hijacker, which is used by `trampoline` to compute the target address and the size of the memory-update operation. In particular, we place an offset/address (`0x28` in our case) as a 64-bits value, the size of the memory update determined at compile-time (if possible; 4 bytes in our case), and packed set of flags which are used to drive the execution of `trampoline`. These flags tell, e.g., whether base/index registers (and which ones) are used, and whether the memory update operation is a string instruction or not.

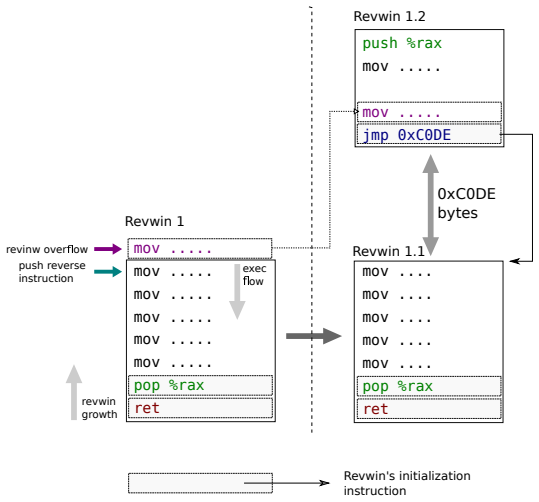


Fig. 15. Undo code block composed of multiple buffers

Since all this information is generated at compile time, and is placed directly on the stack, we rely on 6 properly-generated `movl` instructions[12]. The destination of these instructions is the top of the stack, which is therefore moved before the actual moves using a `sub $0x80, %rsp` instruction. This is an arithmetic operation, and therefore might clobber the x86 `FLAGS` register. We have no guarantee—at least, not without more complex binary analysis covering all possible execution flows—that the memory-update operation which triggered the generation of this code does not come after any `cmp` instruction. Therefore, we cannot ensure that the content of `FLAGS` will not be used by any conditional instruction after the `movsd` instruction is executed. To this end, we save the content of `FLAGS`. This is represented in the picture by the `pushfw` instruction for visual simplicity. The actual code that we inject, on the other hand, is as follows:

```
pushq %rax
lahf
seto %al
pushq %rax
```

This is a semantically-equivalent snippet to save arithmetic-instruction-related flags which, differently from a `pushfw` instruction, avoids a possible flush of the X86 pipeline, thus reducing an additional cost—this is despite the fact that more instructions are actually executed.

Once the call to `trampoline` returns, the instruction to reverse the execution of `movsd` has already been generated[13]. Yet, before resuming the normal execution of the instrumented software, we still have to undo the effects of the previous injected instructions. This is done using an `add $0x80, %rsp` and a (logical) `popfw`.

A.2. Managing Variable Size Undo Code Blocks

As we have mentioned in Section 3.2, undo code blocks are stored into memory buffers served from the heap using a Linux-like slab allocator. This means that each buffer

[12]We do not use `movq` instructions to make the code backwards compatible with 32-bit systems, and since `movq`s do not allow 64-bit immediate operands on 64-bit systems.

[13]We note that in this case, the corresponding reverse instruction is a simple `movl` instruction, which simply copies the bitmask corresponding to the old floating point value. This means that the generation of reverse instructions is *agnostic* with respect to the actual datatype.

has a fixed size. Nevertheless, the actual number (and size) of the reverse instructions generated at runtime is not know beforehand. To account for this, we have adopted two different strategies.

The first one is as well related to the way undo code blocks are executed. In fact, as already mentioned, we place at the very end of this buffer a `ret` instruction, and we push newly-generated instructed using the buffer in a stack-like fashion. In this way, if the size of all generated instructions is smaller than the size of one buffer delivered by the slab allocator, the undo of an event simply entails calling into the first instruction in the undo code block. This is correct, at the cost of wasting some memory from that buffer.

On the other hand, if the size of runtime-generated instructions is larger than the size of one buffer, two different buffers must be connected together. This can be easily done by requesting to the slab allocator a new buffer, yet we do not place a `ret` instruction at the very end of the newly-allocated one. Rather, we place a `jmp` instruction, having as the destination address the one associated with the first instruction stored into the previous buffer. The final organization of a multiple-buffered undo code block is reported in Figure 15, where we show that the actual destination of the `jmp` instruction is computed as an offset to be x86-compliant.

A.3. Details on the Interaction with the Memory Map Manager

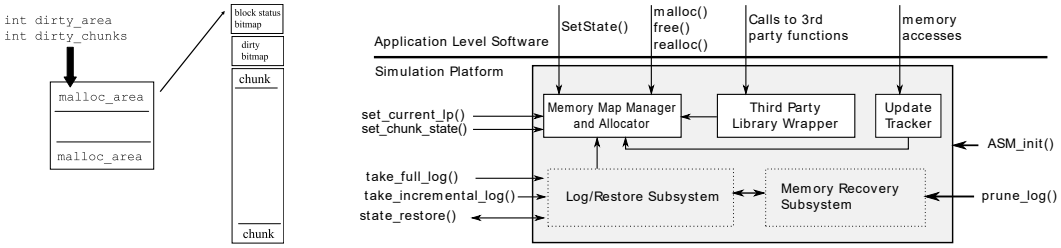
Another point which deserves a discussion, is the way our rollback scheme interacts with memory management. In fact, being able to undo the effects on the content of a memory buffer might not be enough to guarantee a correct rollback operations: the *state* of the buffer must be properly managed as well. In fact, if a generic simulation is allowed to rely on dynamic memory, the forward execution of an event might allocate new memory (via, e.g., a `malloc()` call) or release buffers which are no longer needed (e.g., via a `free()` call). If events involving such calls are undone, the undo operation must correctly restore the allocation state of buffers.

While releasing a buffer to undo an allocation can be trivial (a corresponding `free()` might be executed), the other way round is more complicated. In fact, a sequence of `malloc()`, `free()`, `malloc()` is never garanteed to deliver the same buffer which was allocated by the first call.

We note, however, that this anomaly is independent of the technique used to undo a portion of the speculative simulation. In fact, event checkpoint-based rollback executions might present similar side effects. In particular, we adopt an approach already studied in the literature [Toccaceli and Quaglia 2008], which is implemented within DyMeLoR [Pellegrini et al. 2015], the memory management subsystem used in our experimentation.

In particular, in DyMeLoR the execution of a `free()` call does not really releases the associated buffer. Rather, it simply clears a bit in a metadata table (see Figure 16(a) for the organization of the memory map within DyMeLoR). In case of a checkpoint-based rollback operation, both the metadata table and the content of the memory chunks are restored. For a complete discussion on the correctness of this approach, we refer the reader to [Toccaceli and Quaglia 2008].

Since in the presented approach we do not only rely on checkpoint-based rollback, rather we also undo in the opposite order the effects of updates on the memory map, it is enough to explicitly interact with DyMeLoR so as to correctly restore its internal state. To this end, we couple our undo code blocks with the `set_chunk_state()` internal API offered by DyMeLoR—Figure 16(b) illustrates the internal organization of DyMeLoR and its API. This is a interface which allows to specify the base address of a memory chunk and a new state (`true` for allocated, `false` for non-allocated).



(a) Data structures (b) Subsystems organization and API

Fig. 16. DyMeLoR data structures and organization

By having this API function, whenever a call to `malloc` is intercepted by DyMeLoR, the memory management subsystem notifies this to the undo code block manager which generated the bytes associated with a call to `set_chunk_state(chunk_addr, false)`. On the other hand, whenever a call to `free` is intercepted, the undo code block will keep a call to `set_chunk_state(chunk_addr, true)`. Given that every memory operation is undone in the inverse order, introducing these two changes to manage the state of buffers is enough to guarantee a correct restore.

A.4. Additional Performance Data

In this section we report additional performance data for an assessment of our undo code block based recoverability technique, and its employment in combination with classical checkpointing. This study complements the one presented in the main body of this article because of the significantly different execution profile of the benchmark application we used. In particular, we run experiments with a model of an in-memory data platform, where a cluster of servers maintain the state of a set of data items, and where transactional requests are delivered at the servers, whose simulation leads to the detection of whether conflicting (read/write or write/write) accesses take place, which gives rise to the abort of the transaction contending in an already locked data item. This kind of models have recently acquired relevance, e.g. as a support for (online) dimensioning/configuring of cloud oriented in-memory data platforms in face of specific data sets and differentiated data access patterns [Di Sanzo et al. 2015].

An illustration of the layout of the state of each LP, which models an individual server, is shown in Figure 17. The state is made up by a table keeping basic metadata (e.g. related to statistics associated with the outcomes of simulated transactions), an array of pointers to buckets, each one keeping the current state of a given data item (associated with a specific key), and an array of pointers to the state of the currently processed transactions.

We simulated a data store with 256 servers, each one managing a data partition made up by 10^4 items, for a total of more than 2.5 million items. Batches of transactional data access requests are delivered to each server by proper simulation events, which are scheduled following an exponential distribution of their timestamps. The transactions may entail accessing the local partition or remote partitions, and the access to remote data partitions leads to cross-LP exchange of simulation events, carrying as payload the set of transactional requests that require access to the remote partition. In our experiments we set the batching factor to 5, and the likelihood of accessing a remote partition to 0.2. Each transactional request may access a reduced number of different data items within the data platform, say 5 to 10. Also, the evolution of the transactions in each batch is modeled by having their accesses processed in

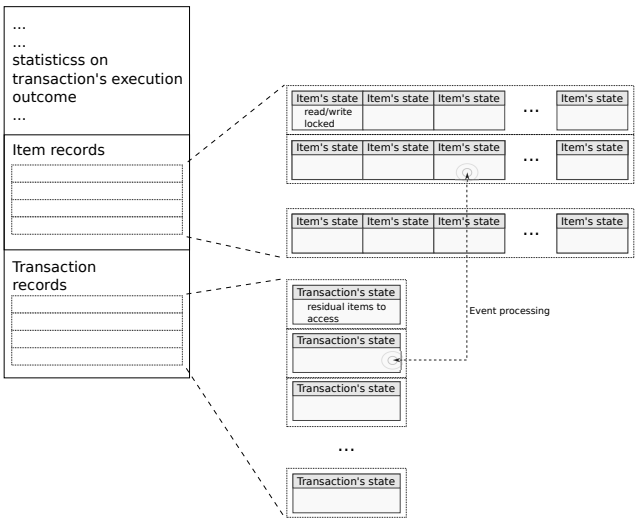


Fig. 17. Layout of the LP state in the data platform model

round-robin fashion. All the parallel runs have been carried out by relying on the same platform already described in Section 4, and by still relying on 32 worker threads.

One interesting point in the settings we used for this benchmark application is that, differently from the case study presented in Section 4, the average latency to take a checkpoint of the whole state of an LP is definitely greater than the average latency for processing and event, in fact the former is of the order of 35 microseconds while the latter is of the order 15 microseconds. Also, the events are write intensive since the access of each item by a transaction leads to updating both the transaction record and the data item record.

We report simulation speed data related to the execution with checkpointing plus coasting forward, with pure reverse scrubbing, with reverse scrubbing used in combination with checkpointing, and finally with the combined usage of coasting forward and reverse scrubbing within a same checkpoint interval (according to the scheme presented in Section 2.3). For the latter configuration we selected 3 different values of the ratio between ν and χ, say 0.75, 0.5 and 0.25.

The results are shown in Figure 18. As expected, for all the curves that relate to the usage of checkpointing, we see a great performance gain when setting larger checkpoint intervals, with no significant performance degradation for very large intervals just due to the reduced event granularity for the case of coasting forward based recovery. Such a fine granularity is also the reason why pure reverse scrubbing is not able to provide the same level of performance of coasting forward. The same is true for reverse scrubbing used in combination with checkpointing. However, the performance curves show the high potential for a combined usage of coasting forward and reverse scrubbing, even in scenarios with fine grain events. In more detail, the combined usage of the two techniques leads to the same (or slightly better) performance when the checkpoint interval is partitioned into equal portions, covered either by coasting forward or reverse scrubbing (say $\frac{\nu}{\chi} = 0.5$). However, we see how a combination of the two techniques that tends to partition the interval in such a way that a larger portion is covered by coasting forward (say $\frac{\nu}{\chi} = 0.25$) leads to up to 8% performance gain, thanks to the avoidance of excessive overhead for the generation of undo code blocks, and the joint possibility to avoid excessively long coasting forwards, which may lead

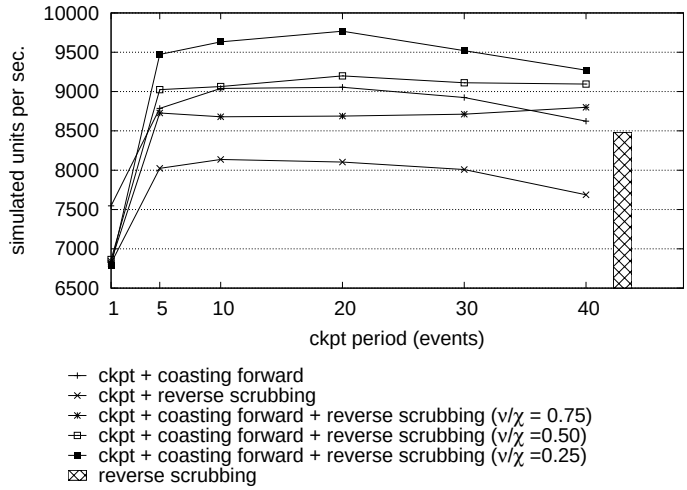


Fig. 18. Results with the data platform model

to non-minimal overhead even with fine grain events. Overall, in synergy with the results reported in Section 4, these curves show how the introduction of the undo code block based technique and its combination with more traditional techniques provides a more efficient support for state recoverability with general workloads, such as fine grain events with write intensive profile as in this tests case.