SAPIENZA
UNIVERSITÀ DI ROMA

A new approach to reversible computing with applications to speculative parallel simulation

Sapienza University of Rome
Ph.D. program in Computer Engineering
XXXI Cycle

Davide Cingolani

Thesis Advisors
Prof. Francesco Quaglia
Dr. Alessandro Pellegrini

Co-Advisor
Dr. Leonardo Querzoni

Reviewers
Prof. Kalyan S. Perumalla
Prof. Christopher D. Carothers

A.Y. 2017/2018

Thesis defended on 22nd February 2019
in front of a Board of Examiners composed by:

Riccardo Torlone (Universitò degli Studi "Roma Tre") (chairman)
Alessandro Farinelli (Università degli Studi di Verona)
Paolo Prinetto (Politecnico di Torino)

A new approach to reversible computing with applications to speculative parallel simulation
Ph.D. thesis. Sapienza – University of Rome

Version: 31st January 2019

Website: http://www.diag.uniroma1.it/∼cingolani/
Author's email: cingolani@diag.uniroma1.it

Abstract

In this thesis, we propose an innovative approach to reversible computing that shifts the focus from the operations to the memory outcome of a generic program. This choice allows us to overcome some typical challenges of "plain" reversible computing. Our methodology is to instrument a generic application with the help of an instrumentation tool, namely Hijacker, which we have redesigned and developed for the purpose. Through compile-time instrumentation, we enhance the program's code to keep track of the memory trace it produces until the end. Regardless of the complexity behind the generation of each computational step of the program, we can build inverse machine instructions just by inspecting the instruction that is attempting to write some value to memory. Therefore from this information, we craft an ad-hoc instruction that conveys this *old* value and the knowledge of where to replace it. This instruction will become part of a more comprehensive structure, namely the *reverse window*. Through this structure, we have sufficient information to cancel all the updates done by the generic program during its execution.

In this writing, we will discuss the structure of the reverse window, as the building block for the whole *reversing framework* we designed and finally realized. Albeit we settle our solution in the specific context of the parallel discrete event simulation (PDES) adopting the Time Warp synchronization protocol, this framework paves the way for further general-purpose development and employment. We also present two additional innovative contributions coming from our innovative reversibility approach, both of them still embrace traditional state saving-based rollback strategy. The first contribution aims to harness the advantages of both the possible approaches. We implement the rollback operation combining state saving together with our reversible support through a mathematical model. This model enables the system to choose in autonomicity the best rollback strategy, by the mutable runtime dynamics of programs. The second contribution explores an orthogonal direction, still related to reversible computing aspects. In particular, we will address the problem of reversing shared libraries. Indeed, leading from their nature, shared objects are visible to the whole system and so does every possible external modification of their code. As a consequence, it is not possible to instrument them without affecting other unaware applications. We propose a different method to deal with the instrumentation of shared objects.

All our innovative proposals have been assessed using the last generation of the open source ROOT-Sim PDES platform, where we integrated our solutions. ROOT-Sim is a C-based package implementing a general purpose simulation environment based on the Time Warp synchronization protocol.

Ringraziamenti

Vorrei esprimere la mia più profonda gratitudine al mio *advisor* ed i miei *tutor* Prof. Francesco Quaglia, Prof. Bruno Ciciani e Dr. Alessandro Pellegrini, per la loro costante guida ed incoraggiamenti durante questi anni di lavoro, studio e condivisione passati insieme. É grazie alla loro passione e all'entusiasmo che hanno sempre profuso nei temi di ricerca e nelle metodologie che ho scelto di perseguire questa tanto perigliosa quanto soddisfacente strada del dottorato e che ha portato alla nascita di un gruppo di lavoro tanto bello, ricco e coeso.

Un ringraziamento doveroso va, dunque, anche a tutti i membri del gruppo HPDCS. Una grande famiglia che, accomunata dalla passione per la ricerca, ha reso speciali questi tre anni di dottorato donandogli qualcosa che va oltre il puro contributo scientifico o di crescita proessionale, comunque presenti, toccando valori importanti quali l'amicizia e l'affetto.

Infine, ma non certo per importanza, vorrei ringraziare davvero di cuore la mia famiglia, i miei parenti e tutti i miei amici. Mi avete sostenuto in ogni momento ed in ogni condizione, mi avete fatto vedere ciò che in quel momento non riuscivo a percepire, mi avete aiutato a rialzarmi dalle innumerevoli "cadute" e non avete mai smesso di credere in me anche quando io stesso avevo difficoltà a farlo. Anche all'infuori di questo progetto siete sempre accanto a me, rendendo ciascuno di voi davvero inestimabile.

Le parole non sempre sono sufficienti ad esprimere tutta l'emozione, tuttavia spero che possiate sentire il mio affetto e vedere il vostro contributo in quest'opera resa possibile grazie alla partecipazione di ciascuno. La conoscenza, l'intuizione, l'entusiasmo e la passione che (spero) qui troverete sono le medesime di cui mi avete fatto dono e per le quali vi sono infinitamente grato.

Acknowledgements

I would like to express my deep gratitude to my research's advisor and co-advisor, Prof. Francesco Quaglia, Prof. Bruno Ciciani and Dr Alessandro Pellegrini for their guidance and encouragement during these years of work, research and sharing. It is thanks to their passion and enthusiasm they always lavished on the research subjects and methodologies that I had no doubt, at the time, to choose diving myself into the as perilous as satisfactory adventure of the PhD. The same driving force which brings a research group so beautiful, rich and cohesive to emerge.

A due thank, hence, goes to all the members of the HPDCS research group which like a big family sharing the passion for study and research makes this three years special, offered something beyond pure scientific or professional contributions, yet present, giving much more important values like friendship and affection.

Finally, though certainly not least, I would like to thank my family, my relatives and all my friends. You supported me at every moment and in every condition, you showed me what I could not perceive at that moment, you helped me to get up from the countless "falls" and you never stopped believing in me even when even I had trouble doing it. Even outside this project, you are always beside me making each of you invaluable.

Often words are not sufficient to thoroughly express the emotions. Nonetheless, I hope you could perceive my fondness e to see your contribution in this writing, made possible thanks to the involvement of each one. The knowledge, intuition, enthusiasm and passion which you will (hopefully) find here are the same you gave me as a gift and for which I'm infinitely grateful.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Basics

The reverse side also has a reverse side.
— Japanese proverb

Many complex systems, from parallel to distributed applications, may need to perform work which is somehow "undo-able", either to prevent unrecoverable system crashes or to untie speculative decisions that may generate infeasible or inconsistent evolution trajectories. These cases are commonly considered as generic "misbehaving" operations, with respect to the application's logic. The advent of multi-core and many-core architectures sharpened this requirement even more. Indeed, handling parallel applications efficiently needs a different design. Multiple and simultaneous execution flows may coexist, requiring to orchestrate complex synchronization and communication aspects. Reversible computing, in its general form, has been employed (and it does continue) as a solution to address rollback-based synchronization issue. Indeed, the most consolidated way to realign the system to a consistent state is to *dismantle* the portions of computation which brought the system to misbehave.

Large-scale parallel computations and distributed systems heavily exploit the available underlying resources, increasing the likelihood to lead the system into some incoherent state. As a consequence, they need a software layer able to synchronize and properly orchestrate the whole system. They pose a few big challenges that currently have been resolved in a non-definitive way: (*a*) fault-tolerance, (*b*) synchronization, (*c*) efficient debugging support and application's profiling, (*d*) energy-efficiency issues. These challenges directly affect systems' dependability which, in its turn, embraces several aspects to ensure: availability, reliability, security and performance. In database and distributed systems, concurrent elements are typically *transactions* which try to access shared resources simultaneously, though even in much simpler architectures conflicts may occur at the level of machine instructions; an example is that of speculative execution, instruction pipelining or branch prediction. Similarly to other systems, also paral-

lel applications introduce a high contention factor to access shared resources. An example is that of speculative parallel simulation applications; they consume millions of events speculatively—the basic idea beneath is to relax synchronization and causality constraints—many of which might be rolled back due to possible inconsistencies generated, e.g. by priority inversions. The widely adopted solution is to *rollback* the whole system to a previously saved coherent state; this technique goes under the name of *state saving*.

In this writing, we primarily focus on the first two challenges; although, we carried out additional research work also addressing the others. The first milestone has been to provide approaches that go beyond classical solutions found in the literature, where generic misbehaviour (e.g. faults, intrusions or causally inconsistent executions due to speculative processing) is supported via *checkpoint-based* rollback, enabled by the state-saving technique mentioned above. In this case, upon any misbehaviour occurrence, the runtime environment rolls back the system's state to the nearest checkpoint, blowing away a (restrained) portion of computation. Once a consistent state has been restored, the system resumes its forward execution flow. Anyhow, with nowadays scalability levels in the underlying hardware platforms, and with the growth of memory-hunger applications (e.g. due to 64-bit based address spaces), saving the system's state is becoming a significant concern both for CPU costs and for direct/indirect memory costs.

As the reader may have already asked himself/herself: what if the system would be able to act also *in reverse*, instead of this rollback&replay mode? Contrarily to state saving, we adopt a backward approach which realigns the whole system to a consistent state by simply rewinding computational steps. Reversible computing is a very appealing alternative to conventional state-saving based rollback. Despite someone can imagine, reversible computing is a rather ancient research field, which only in the latest years has gained a growing attention, both for energy efficiency aspects and for software performance. Reversible computing may have several potential applications in (*a*) computer security, (*b*) transaction processing, (*c*) intrusion detection systems, (*d*) debugging activities, (*e*) developing aiding tools (i.e. IDE), (*f*) and backtracking reasoning.

Objectives of the thesis are to develop a broad reverse-enabling support that comprises an on-line reverse code generator and a set of algorithms and models to embed our reversible computing rollback strategy next to traditional state saving. This hybrid interweaving of rollback strategies is the base of our autonomic system that tailors reversibility support to application's runtime dynamics. The overall framework relies on a *static* binary instrumentation tool (namely Hijacker, refer to Section 3.2 "*Addressing code instrumentation*") in order to integrate our reverse code generator module in target applications. Through instrumentation

we are able to support transformation of generic applications into reversible ones in a *transparent* and automatic way.

Our case of study is that of speculative simulation platforms, and more in details the speculative Parallel Discrete Event Simulation (PDES). From a generic point of view simulation is a problem-solving technique to cope with complex models generally conceived from real (or hypothetical) phenomena, which are non-trivially reproducible otherwise. Simulation applications handle a considerable number of parallel/distributed objects interacting together by message passing. Each object is a logical entity which lives into a *virtual time* context unrelated to wall-clock time, and consumes incoming messages. Event-based simulation enforces the equivalence between those messages exchanged and the relative triggered events. Unlike other structured processes, speculative simulation adopts optimistic heuristics, which allow to perform scheduled events even if they are not *safely* processable—the safety property straightway depends on the actual processing order of events with respect to global causality relationship they are involved in. The optimistic approach looses event processing constraints and exploits computational resources much better. Nevertheless, it might lead the system to violate the causal order, bringing the simulation to an inconsistent state. The more complex the simulation model runtime dynamics is, the more likely it requires to rollback out-of-order events. So far, the most consolidated way to rollback is to employ state saving techniques, though it exhibits considerable memory overhead and time latency as the simulation model gets more complex. Leading from the previous considerations, it appears clear that simulation models usually require a considerable storage allocation to model the reality and to store partial results, which further observe several updates per time. Nonetheless, the optimistic (speculative) slant increases the likelihood of rollback operations. Simulation applications exhibit also CPU-intensive loads, making them a representative testbed to assess our approach in a wide spectrum of cases at once.

The key innovation that distinguishes our work from the solutions proposed in the literature is basically the viewpoint change in the data structures and logic supporting the rollback operation. Traditional solutions require to maintain a complex data structure that has to hold some meta-data, beyond the real data to restore. Meta-data builds the proper application's memory layout and, consequently, the knowledge of how data will have to be restored. Rather, we shift the focus from data to *instructions*. This new approach harnesses the intrinsic power of machine instructions to convey both data and the relative restore logic together. It results in a single complete and yet compact structure, namely the *reverse window* (refer to Chapter 3 "*The reversing framework*"). This approach reduces notably the runtime overhead required for saving the application's state and generating

the necessary restore-enabling meta-data. Furthermore, the rollback operation becomes independent of the forward-operation complexity; since we climbed the abstraction level down to the machine instructions, this enables to leverage a raw yet uncluttered outline of the memory usage.

1.1 Motivation

Reversible computers are the future alternative that will eventually replace current conventional architectures. The growing demand for more energy-efficient hardware and the impossibility to trespass an insurmountable physical limit in the manufacturing process of transistors have now become evident. Such a reversible hardware architecture, though, does not exist yet but as a proof of concept. This topic has a profound ground in physical and mechanical aspects, and an increasing number of researchers are investigating new technologies to bring these architectures to reality. Other research branches, on the contrary, place themselves at a higher abstraction level to respond orthogonally to the need that reversible computing can address. This is the rationale that motivated us in realizing a thorough reverse-transforming framework (i.e. `libreverse`, Chapter 3 "*The reversing framework*"), which will be the core of this thesis. Nevertheless, we would like to stress that, even though software-reversible computing is our context, this is not unrelated to its physical counterpart, at least in principle. This is also a reason why in Chapter 2 "*State of the Art*", dedicated to provide the background of the related work, we will deserve a digression on the general reversible-computing theory.

Settling in the broad context of reversible computing, we aim to provide solutions from a *software* perspective, targeting the rollback-enabling supports for those applications which require to undo generic misbehaving operations. Going beyond what it has been already done in the literature, we conceived an innovative way to employ reversible computing through a general-purpose framework that enables for a reverse-transformation of any generic application into a reversible version (see Section 3.2 "*Addressing code instrumentation*"). Although we focus on PDES, applications of that framework span over different fields, from debugging and profiling systems, to efficient rollback paradigms to address synchronization in parallel execution. The scope of this work is to describe and evaluate the contribution's impact of the innovative approach we began to early develop in [27], where we start addressing challenges that conventional state-saving rollback approach solves in a non-definitive way. While devising such a high-level framework, we bear in mind some fundamental objectives to pursue: (*a*) efficiency, (*b*) transparency, (*c*) autonomicity and (*d*) portability. As for the first *efficiency* objective, the search for improvements of runtime performance is the driving requirement of this work. Further on, to meet the developer's needs of focusing

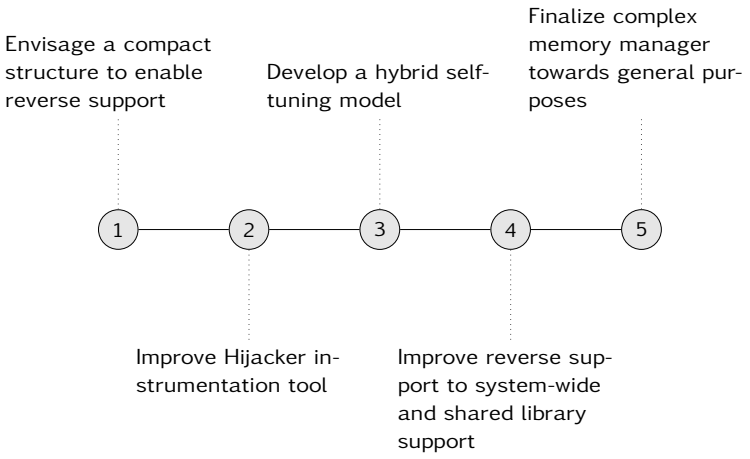


Figure 1.1. Roadmap of the work done.

on his/her work and not burdening he/she with the additional encumbrance of handling reverse-related stuff, *transparency* arises as a fundamental requirement. Provided that not every developer has the knowledge and/or the sufficient ability to design reversible code properly and devise an optimized version of it—him/her may be a domain expert not a programmer,—the reverse-enabling framework must provide a complete toolset to take care of that. Analogously, every application has its own characteristics and peculiarities; hence it is nearly impossible to find a "one size fit all" solution to achieve the best performance in every context. This observation is the main motivation behind the *autonomicity* requirement. It somehow catches a problem similar to transparency, but it further enables the system to better suit dynamic application's behaviour under its actual contexts and/or runtime conditions. Besides the objectives mentioned above, we also consider *portability* to enable for an "out of the box" pluggable support. For this reason, we also underline the broad scope our reverse-enabling support framework aims.

From these assumptions, it was born `libreverse`, one of the primary efforts of this work. It is a comprehensive library providing broad reversing support, capable of being plugged into any generic software, both user-level and system-level. `libreverse` is part of what we call the *reversing framework* which, indeed, comprises also a set of others tools we devised for the purpose. As a result, the whole reversing framework is a sort of history and memory manager, providing the best reversibility support strategy, according to the application's characteristics.

Figure 1.1 gives the roadmap overview of the effort spent on developing a comprehensive reverse-enabling framework, from the very beginning of envisaging an efficient data structure able to maintain all the meta-data necessary to the upper level reversing function and the reversing function itself, until to devise and integrate toolset to enable for an "out of the box" usage.

As a last introductory note, we remark that the content of this thesis appeared also in the following list of publications:

- Cingolani, D., Pellegrini, A., Schordan, M., Quaglia, F., & Jefferson, D. R. (2017). Dealing with Reversibility of Shared Libraries in PDES. In Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (pp. 41–52). https://doi.org/10.1145/3064911.3064927

- Cingolani, D., Pellegrini, A., & Quaglia, F. (2017). Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. ACM Transactions on Modeling and Computer Simulation, 27(2), 26. https://doi.org/10.1145/2769458.2769482

- Cingolani, D., Ianni, M., Pellegrini, A., & Quaglia, F. (2016). Mixing hardware and software reversibility for speculative parallel discrete event simulation. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 9720, pp. 137–152). https://doi.org/10.1007/978-3-319-40578-0_9

- Cingolani, D., Pellegrini, A., & Quaglia, F. (2015). RAMSES: Reversibility-based agent modeling and simulation environment with speculation-support. Lecture Notes in Computer Science (Vol. 9523). https://doi.org/10.1007/978-3-319-27308-2_38

- Cingolani, D., Pellegrini, A., & Quaglia, F. (2015). Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. In Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (pp. 211–222). New York, New York, USA: ACM Press. https://doi.org/10.1145/2769458.2769482

The research effort spent while working on this thesis has also contributed to the following publications:

- Ianni, M., Marotta, R., Cingolani, D., Pellegrini, A., & Quaglia, F. (2018). The Ultimate Share-Everything PDES System. In F. Quaglia, A. Pellegrini, & G. K. Theodoropoulos (Eds.), Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Rome, Italy, May 23-25, 2018 (pp. 73–84). ACM. https://doi.org/10.1145/3200921.3200931

- Conoci, S., Cingolani, D., Di Sanzo, P., Pellegrini, A., Ciciani, B., & Quaglia, F. (2018). A Power Cap Oriented Time Warp Architecture. Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, (i), 97–100. https://doi.org/10.1145/3200921.3200930

• Economo, S., Cingolani, D., Pellegrini, A., & Quaglia, F. (2016). Configurable and efficient memory access tracing via selective expression-based x86 binary instrumentation. In Proceedings - 2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016 (pp. 261–270). IEEE Computer Society. https://doi.org/10.1109/MASCOTS.2016.69

1.2 Thinking in "reverse"

Albeit a more detailed discussion about reversible computing will be addressed in Section 2.1 "*Reversible computing*", we would like to introduce the topic to give the reader a prompt overview of the context. We shall proceed by giving a pragmatic definition of reversibility:

Definition. Let A be an automaton. It is reversible if the transition function from a generic state S_k to another generic state S_n is invertible.

Reversible Computing refers to computational models performing *logically* reversible operations. That is, instructions which allow to rebuild any previous state by relying only on the knowledge of the current one, stepping backwards through the same computational steps already performed in forward execution. The fundamental requisites are the following:

- No information has to be lost (or implicitly erased) during the operation.
- The transition function, from a generic state to the following one, must be bijective.

If the former requirement is more related to the physical realization of digital logic and circuitry, the latter regards software implementation of the algorithms. This already gives an insight on the underlying strict correlation between physical and logical aspects. Further, reversible computing has to face some big challenges that prompted several researchers. Clearly, *perfect* physical-reversible architectures are infeasible; quantum theory clearly explains that any environment has always a non-zero possibility to be irreversibly altered by the everlasting Universe's ground noise which creates entropy. Notwithstanding that, interesting attempts have been done (see Section 2.1). Vice versa, no existing software implementation is generally designed to be *natively* fully reversible; any application always exhibits a certain degree of irreversibility, as a leading inheritance of the programming languages' structure and underneath computing architectures. Therefore, one of the biggest hurdles in achieving software reversibility is indeed overcom-

```
1 percent(base, perc) {
2     int result = base;
3     result *= perc;
4     result /= 100;
5     return result;
6 }
```

Listing (1.1) Forward version of the function `percent()` that calculates the percentage.

```
1 percent_i(value, perc) {
2     int result = value;
3     result *= 100;
4     result /= perc;
5     return result;
6 }
```

Listing (1.2) Reverse version of the function `percent()`.

ing the forward-determinism of our traditional computational models, and thus of their relative programming languages.

Settling in this context, the effort of this thesis work is to propose solutions at the *software* level, targeting those applications that require to undo misbehaving operations from a high level logical perspective.

Unlike the traditional concept of reversible computing (or reversible computation), we embrace a slightly different form of reversibility in this work. To be more precise, the more general reversible computation in computer theory addresses how to reverse a generic computational step s_k. The objective is to build an inverse function F^{-1}, provided the generic forward counterpart F composed of several computational steps s_k $\forall k \in \mathbb{N}$. Our approach, instead, changes the focus only on the outcome of that computation "perceived" by memory, rather than on the single operations done by our example's function F. In this way, we can overcome some challenges that are still open in the reversible computing theory, and simplify the creation process of the inverse code block—we will call it *undo* of *reverse code block*.

1.3 Discrete event simulation

To proceed further on this dissertation, it is worth to spend few words to introduce the specific application context this work has taken as "use case". As a first step, we focused on discrete event-driven simulation (DES) applications and in particular its parallel version (PDES). This section is dedicated to introduce the simulation world, which happens to be the case study of this research work. Let's introduce simulation applications by first answering to the following questions: why simulation? and why discrete event, specifically?

Answering to these questions is fairly simple. Simulation is a proven technique able to capture a lot of physical phenomena whose testing in a real envi-

ronment would be infeasible or, at least, impractical. Hence, the straightforward idea is to create an "in-vitro" environment that captures every interesting aspects of the real one, except for being scaled down to a manageable size. The reason behind the adoption of discrete event simulation is related to its architecture; it often handles a high number of distributed objects interacting together by message passing. Since simulation relies on a *Simulation platform!virtual time* concept, the correctness of messages causality must be ensured. Unlike other structured processes, event-based simulation systems may adopt speculative exploration/execution, which could bring the system to inconsistent simulation states. This is the typical case of optimistic heuristics, which allow to handle scheduled events even if it is not yet proven that they are *safe* to be actually executed. This speculative approach loosens some causal constraints and better exploits computational resources, at the expenses of a burden to handle "misbehaving" cases. If on the one hand the solution allows to gain efficiency and guarantees to avoid any kind of deadlock, on the other hand many times, operating entities are faced with the following dilemma: whether to follow one branch or the other. The answer adopted is to speculatively perform one of the two by choosing according to some criterion, even though it may not be the right decision.Taking a checkpoint before a branch is followed, allows to undo the mis-predicted portion of computation and to bring the system back to a previous consistent state.

Event though discrete event simulation is a specific context, it is representative of a more generic kind of applications where the "rollbackable" execution is an endemic phenomenon in their runtime evolution. As mentioned, the rollback operation is a fundamental building block to support the correct execution of speculative systems, such as Time Warp-based ones [65]. In the literature, several solutions to reduce the execution cost of this operation have been proposed, either based on the creation of a checkpoint of previous simulation state images, or by executing negative copies of simulation events which are able to undo the updates on the state. State saving is a widely employed, simple and easy-to-implement solution that allows for a reversible execution; nonetheless it may consume much memory to save a program's history snapshot.

Differently from other proposals, we address the issue of executing backward updates in a fully-transparent and event granularity-independent way, by relying on static software instrumentation (targeting the x86 architecture and Linux systems) to generate at runtime *reverse code blocks* (or *undo code blocks*)—which differ from reverse-events implemented as pure inverse shadow copies of the event itself, "proper" of the reverse computing technique.

So far in the literature (see also Chapter 2 "*State of the Art*") we have seen two approaches to build a reversible support: (*a*) saving copies of variables to memory or (*b*) computing inverses of individual operations without relying on memory at

all. Although the former approach is simpler to implement, it introduces a time overhead due to additional instructions in charge of saving values, and moreover it may require an exceeding memory amount to store old copies. Further, memory-based reversal might also negatively affect the memory subsystem behaviour (in terms of cache misses and TLBs) during snapshot re-installation. Conversely, performing inverse computations allows to consistently reduce memory costs, as no additional storage latency is required and no additional processing time would be wasted for memory movement operations.

Memory latency represents a limit of current undo schemes which actually checkpoint-based solutions heavily rely on. The pursuit was to relieve memory pressure in rollback operations by focusing on machine instructions, rather than to dump a composite and (relatively) large structure—in other words, instead of taking periodic snapshots of the application's state, the instructions' outcome would be tracked. To this end, the challenge to deal with was the optimization of the ratio between forward and backward instructions. The key objective in this first research's phase has been to optimize memory requirements of reversible computing support's needed to implement rollback facilities, from two orthogonal perspectives: (*a*) the pure storage size, and (*b*) the number of memory accesses.

Speculative processing is proven to be very effective, nonetheless its current implementation still lacks optimality guarantees. As we have seen, traditional approaches try to provide for it by varying either the frequency at which checkpoints are taken or by devising delta-based schemes to build incremental snapshots. All these solutions, though, do not ensure to achieve straightforward the *optimum*, which requires a fine tuning process to best match application's characteristics. This is an operation that results in tuning parameter's values, that cannot be easily computed a-priori and could be time-expensive to determine. To better understand this aspect, it is sufficient to observe that each application has a different nature which reflects itself in the rollback characteristics. For example, fault-tolerant applications will have a reduced rollback ratio, whereas speculative-processing-based ones would likely show a much higher degree of event's concurrency, thus of consistency violation probability, especially in adverse runtime scenarios. Applications themselves rarely have a homogeneous behaviour, rather each of them shows a different trajectory footprint which reflects distinct computational phases. Each phase likely evolves showing a certain events' processing density, therefore it is characterized by different rollback frequency. Such a difference in processing density is reflected in a non-balanced advancement of the components along the simulation time.

Leading from these considerations, the optimum rollback strategy is not unique and quite often hard to compute in advance, moreover it is likely not able to suit an application throughout its whole extension. To this end, a supplementary con-

tribution we focused on in Chapter 4 "*The hybrid rollback strategy*" is to provide *autonomicity*, by investigating hybrid and yet transparent approaches. The devised algorithms will dynamically choose the best reversibility scheme according to the specific application or execution context, blending different schemes together under a single composite strategy. The result is transparently tailored to the software's features, hence able to better catch its runtime execution's dynamics.

Chapter 2

State of the Art

Any sufficiently advanced technology is indistinguishable from magic.
— Arthur C. Clarke, Profiles of the Future: An Inquiry Into the Limits of the Possible

To make unfamiliar readers comfortable with the context of reversible computing, we believe it is worth to deserve a chapter to a deeper insight into the history behind it. Reversible computing is a vast and rather ancient research field which stirred several researchers to investigate what is the linkage between information theory and its corresponding physical representation. It interweaves several perspectives, from mathematics to physics, electronics and informatics. Within this broad context, and throughout the thesis, we identify two main abstraction levels: (*a*) physics and (*b*) software. They provide two orthogonal yet complementary perspectives. The former clearly regards the proper physical substratum, inspecting the correlation between the entropy generation and the information computing theory. Indeed, reversible computing is historically steered to the physical perspective of computing platforms, investigating a new dimension of computational models and practical technologies (e.g. cellular automata, adiabatic circuits or billiard-ball computational models) to build an "entropy-less" reversible computer. Notwithstanding, reversible computing has widened its perspective touching the software layer. Software reversible computing regards techniques and algorithms developed to overcome the intrinsic forward-determinism proper of the conventional computer hardware architectures and make a program reversible. Applications of software reversibility span to:

- synchronization;
- fault tolerance;
- application security sandboxing; *and*
- debugging and profiling.

In Chapter 3 "*The reversing framework*", we propose a solution to address the first of the bullets mentioned above, namely *synchronization*. The following Section 2.1 is dedicated to providing a more detailed overview of the reversible computing domain, notwithstanding it is not strictly preparatory to our scopes. On the contrary, Section 2.2 gives the related work overview set in the specific context we embrace, namely Parallel Discrete Event Simulation (PDES). A comprehensive work aimed to who is approaching to the reversible computing realm is the book "Introduction to Reversible Computing" of Kalyan Perumalla [88]

2.1 Reversible computing

One of the toughest challenges the semiconductor industry is currently facing can be summarized by the so-called "dark-silicon". According to the well-known Moore's Law, transistors were made continuously smaller, becoming simultaneously cheaper, faster, and more energy-efficient. However, multiple issues appeared in this winning scenario. The main obstacle to this trend regards a feasibility limit of the fabrication processes of transistors and integrated circuits. Thus, when short-channel effects resulted in approaching a plateau of clock speeds, the industry shifted to multi- and many-core architectures. Notwithstanding, even these architectures begun to suffer. Heat dissipation constraints prevent to power all the possible cores available, bounding the full computational power they could provide. We are approaching a clear untraversable point, beyond which Physics tells us we cannot push ourselves much farther because of the entropy generation, resulting in heat. Reversible computing paves the way for a new promising dimension, where we may find solutions to overcome these limitations [51].

What is "reverse computing"? In [78], Kenichi Morita gives a plain definition of reversible computing. We quote:

> *Reversible computing is a paradigm that has a close relation to physical reversibility.*

By far, this definition seems rather generic and even too much simplistic, however it clearly catches the essence of this research branch. Reversible computing grounds its roots in the past [123, 70]. Indeed, starting from Rolf Landauer who in the 60s first argued (Theorem 2.1) the relation between computing reversibility and physical reversibility, other researchers began to investigate the implication:

$$Logical\ reversibility \iff Physical\ reversibility \tag{2.1}$$

In the literature, reversible computing refers to computational models founding on reversible physical phenomena and reversible (logic) operations. Conventional computers are founded on intrinsically irreversible computational models. The motivation behind so much effort in reversible computing, on the contrary, is that it represents an entirely new dimension of reasoning, without information loss. Historically, the main motivation behind reversible computing has been energy efficiency, due to its profound relation with physics. Nowadays, it is continuously gaining ever-growing attention both for energy efficiency aspects and for performance (e.g. scalability of concurrent applications). Further, also other perspectives have been addressed beyond theoretical physical implications or logic circuit design—we refer to the engineering of reversible hardware supports,—but also in the higher level of abstractions. At the software level, reversible computing is used as an adjective for algorithms and programs able to perform some backward deterministic operation, though still on top of traditional information-loss hardware.

The concept of reversible computing grounds on the combination of traditional *forward* determinism and its *backward* counterpart. In other words, a generic application or program can be considered "reversible" if by no means there exists a pair of computational states that can reach two different final states, provided the same execution context. Fundamental requisites for reversible computing are the following:

a. No information has to be lost during computation;

b. the transition function, from a generic state to the following one, must be bijective.

To give first the concept behind its potentialities, we intentionally introduce the Definition 2.1 of what reverse computation *allows* to do, instead of what it actually *is*. This choice is also preparatory to understand how to achieved reversibility in generic programs.

Definition 2.1 (*Reversibility*). A program is defined to be reversible if it allows to backward retracing all the computational steps between the current state and any previous one. In other words, reversibility implies that each state S_k has a unique predecessor S_i and, thus, a unique backward execution trajectory.

$$\begin{aligned} &F(S_i) \implies S_k \\ &F^{-1}(S_k) \implies S_i \qquad\qquad S_i \prec S_k \end{aligned} \tag{2.2}$$

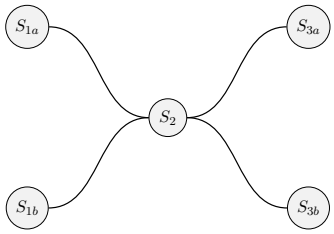


Figure 2.1. Non-determinism in traditional computational models.

where $F(\cdot)$ represents the generic transition function from one computational state S to another and $F^{-1}(\cdot)$ is its inverse. The $\prec$ notation introduce a total order between the two computational states, so as S_i is a predecessor of S_k.

From a practical point of view, we define a program as *reversible* if it is possible to pause its forward execution at any time and walk backwards throughout the same computational steps, and vice versa. Figure 2.1 explains, by means of a counterexample, the underlying concept of bi-directional determinism at the foundation of reversible computing. The picture represents the computational steps of a generic program, for a given input. Let's assume to be in state S_2, ignoring for the moment how we have reached it. In this computational state, let's suppose to pause the forward execution to step back to our previous state; where should we go? It is clear we have no sufficient information to decide whether the former state is S_1a or S_1b. Analogously, if we decide to let the forward execution proceed, we again do not have a deterministic behaviour of the transition function, which has two edges; one toward S_{3a} and S_{3b} the other. In other words, there exist two subsequent runs of the program such that they may produce a different final output, provided the same input.

The remainder of this section is dedicated to an overview of the broad context of reversible computing. Section 2.1.1 "*Physical perspective*" provides some additional details on physical implications of reversible computing through an excursus of the research's evolution. Albeit interesting, this dissertation does not hinges on the physical perspective, rather on a higher level of abstraction. Therefore, Section 2.1.3 "*Software perspective*" introduces reversibility in software, which is the context of this work.

2.1.1 Physical perspective

Despite the great interest and important involvement that physical aspects have concerning computation and computer theory, this matter is slightly out of the scope of this thesis. However, it is worth to spend some words on this topic to give

the reader a broader viewpoint of the context underneath the software level we worked on and will present in the remainder of this dissertation.

As we think about reversibility, probably our mind first goes to physics and its natural phenomena, most of which are actually reversible. It is sufficient to think about simple processes like water freezing and melting, or the gas expansion within a piston chamber which can be compressed and let be expanded. However, if we think about a piece of wood burning, the process does no longer seem to be reversible. Indeed, it is not reversible because fire introduces entropy, which eventually translates into heat. If now we come back to computing architectures, the viewpoint is much more similar to the latter example. It is due to our hardware design, built on top of an irreversible process: the bit flipping. To be precise, the action of flipping one bit from a generic logical state to the other (e.g. from one to zero) is not irreversible per se; however, it implies the consumption of a quantum of energy and the irrecoverable erasure of the previous state. The intrinsic reversibility of physics' nature prevents we can ever truly erase information in a computer; rather, it "moves" to the machine's thermal environment turning itself into *information entropy* [106], that external world perceives as heat. Erasure, in this case, exists only from a logical perspective. Recalling the example in Figure 2.1, the information entropy reflects into the non-determinism that prevents to move backward through a single edge

Definition 2.2 (*Physical reversibility*). Processes are defined to be physically reversible if they do not dissipate energy in heat, hence without generating entropy.

People early start asking themselves what would be the relation between the physical components of the underlying computing architectures and the upper application's logic. In other words, what is the relationship among Definition 2.2 and the following Definition 2.3.

Definition 2.3 (*Logical reversibility*). An operation is logically reversible if its computational state, just before it is carried out, is uniquely determined by the output state. Therefore, no information must be erased, or otherwise lost.

The direct consequence is that a logical reversible program is backward deterministic along its trajectory.

Recall the operation of changing the bit status and consider a real system. If it would be possible to flip bits reversibly, hence without destroying information, then how much it would be the minimum amount of energy dissipated? To answer this question the researcher Rolf Landauer, in a paper published in 1961 by IBM [70], argued the implications of the logically irreversible nature of the

conventional computational operations on the thermodynamic behaviour of the underneath device. Landauer's reasoning can be understood by observing that the most fundamental laws of physics are reversible, meaning that if you had complete knowledge of the state of a closed system at some time, you could always run the laws of physics in reverse and determine the system's exact state at any previous time, at least in principle. Theorem 2.1 expresses what goes under the name of the Landauer's principle. It states the relation between heat dissipation and the "erasure" of information bits. In his essay, Landauer also argues the inevitable presence of logic (and physical) irreversible operations in traditional computational models.

Theorem 2.1 (*Landauer's principle***).** The irreversible loss of 1 bit of computational information requires the dissipation of the following quantity of energy:

$$E_{min} = k_B T_a \ln 2 \tag{2.3}$$

where k_B is the Boltzmann constant ($\sim 1.38 \cdot 10^{-}23$ J/K) and T_a is the temperature of the circuit in which the lost bit finally ends up.

According to Landauer, E_{min} is hence the corresponding minimum amount of energy that any computer will dissipate as heat for every single bit of information erasure[1], which happens to be a 17-thousandths of an electron volt (eV). This result has also been proven experimentally in [22]. However, it is quite interesting to note that current CMOS technology-based conventional computers can dissipate over 5000 eV per bit erasure; much worse than Landauer stated. Efforts in the optimization of this hardware can be made, but it seems not possible to push energy efficiency beyond the threshold of around 500 eV; an order of magnitude less than the previous result, but still far away from the theoretical lower bound identified by Landauer.

Landauer's effort went to prove his approach of reusing the same energy associated with the information-bearing signals for multiple operations, rather than a single one; only then throwing away data. To prove his theory, Landauer ended up to convince himself of the inevitability of information erasure, because of the limited memory available which eventually had required to clean up some previous value. Later on, his colleague, Charles H. Bennet, first showed in 1973 in [18, 19] the theoretical feasibility of logically reversible computers, arguing that it is practical to make any irreversible computation reversible. Bennett, though, addressed only the logical and theoretical aspects, leaving open the problem of how to precisely embed such a reversibility property in real and complex hardware architectures. However, his work was hinged to overcome Landauer's claim that irreversibility is a certain computing property. Bennett built a 3-tapes reversible Tur-

[1] About $3 \cdot 10^{-}21$ joule at room temperature (21°)

ing Machine which proved the reversibility of general-purpose computers, without an overburdening complexity. This is the first attempt in the reversible computing models direction. Bennett's reversible machine is based on the *state saving* technique—basically, he save all the information that would be otherwise lost—therefore, it may require a considerable amount of storage. This fact renders a real implementation in complex systems infeasible, or at least impractical.

The debate regarding the physical implications of reversibility in computing theory is rather heated, however. The discussion regarding physical implications of computing theory is more complicated than it appears and embraces conflicting opinions related to the connection between the Landauer's principle and the second law of thermodynamics. On the one hand, we find supporters arguing the whole groundlessness of the Landauer's principle [69], and, on the other hand, there are supporters, such as Bennet [21], who still support the pedagogic power of the principle in understanding some aspects of computational, nonetheless admitting its limitations concerning practical implications:

> *I would nevertheless argue that Landauer's principle serves an important pedagogic purpose of helping students avoid a misconception that many people fell into during the twentieth century, including giants [...]. Landauer's principle, while perhaps obvious in retrospect, makes it clear that information processing and acquisition have no intrinsic, irreducible thermodynamic cost whereas the seemingly humble act of information destruction does have a cost, exactly sufficient to save the Second Law from the [Maxwell's] Demon.*

As a result of the Bennet's work, reversible computation could, in principle, circumvent the Landauer's law provided it would be implemented onto the right hardware, which though is still missing.

In this direction, the first notable attempts to overcome traditional circuitry architectures was made by Fredkin and Toffoli between the '70s and '80s [53, 52, 50]. They investigated the possibility to build a complex reversible computer on top of a *conservative* hardware architecture, and realized a working hardware prototype of a physical reversible logic gate; namely, the so-called *Fredkin gate* and the *Toffoli gate*. These logic gates are essentially Boolean functions which are both invertible and conservative. The former property regards the mathematical aspect related to the ability of finding an invertible function, whereas the latter refers to the real information conservation throughout the elaboration process. At the same time, also other researchers investigated the possibility to build a reversible architecture, approaching what they called "adiabatic" circuits. The research in this direction cre-

ated a fertile ground for a subsequent generation of researchers who spent more effort on that branch; among others Michel P. Frank at MIT where he graduated with extensive work on this topic [48]. Soon, he and Carlin Vieri built a fully-reversible processor [112, 113, 11], which represents a proof of concept that computer architecture theory could actually be transferred over the reversible computing realm without barriers. In [51], Michel P. Frank comments reversible computing with the following words:

> *To be clear, reversible computing is by no means easy. Indeed, the engineering hurdles are enormous. [...] I would guess that the total cost of all of the new investments in education, research, and development that will be required in the coming decades will most likely run well up into the billions of dollars. It's a future-computing moon shot.*

More recently, a work [41] on the adiabatic computational model has been proposed addressing also security aspects of computation.

2.1.2 Reversible computational models

Usually, digital computers perform operations that erase information along the execution history. Therefore, the immediate predecessor of a computational state could be ambiguous. In other words, traditional general-purpose automatons lack a single-valued inverse function; thus they are not backward deterministic. Starting from the first historical Landauer's claim breakthrough, several attempts were made to prove the general reversibility of computational theory, therefore the equivalence between *physical* and *logical* operations. Then, after that Bennet overturned the belief that information erasure is an unavoidable consequence of the computation process, other researchers have spent effort in studying reversible computation and its physical implications and in exploring possible architectural models to efficiently perform computation in reversible computers.

Reversible Turing Machines In 1973, Bennett [18] first defined the universal reversible computation model showing the feasibility of building a Reversible Turing Machine (RTM). He built a 3-tapes (reversible) Turing Machine, which proved the reversibility of general-purpose computers. A Reversible Turing Machine is described by the same tuple of the traditional forward-deterministic Turing Machine: $\langle Q, \sigma, b, \delta, q_s, q_f \rangle$, where:

Q	Is the set of possible computational states.
σ	Is the set of possible symbols.
δ	Is the transition function $\delta : Q \times \sigma \implies Q \times \sigma \times \{\leftarrow, \rightarrow, \circ\}$.
q_s	Represents the starting computational state.
q_f	Represents the final computational state.

The set $\{\leftarrow, \rightarrow, \circ\}$ describes the possible moves of the needle on the tape, supposing that it is infinite and readable in both directions. Respectively $\leftarrow$ indicates a step backward of one character, the symbol $\rightarrow$ indicates a step foreword of one character and $\circ$ indicates the needle to remain in the same position.

Definition 2.4 (*Reversible Turing Machine - RTM***).** A Reversible Turing Machine is a Turing Machine where the δ transition function is partial bijective. Since the transition function is a partial function, partial bijection relaxation implies that it is sufficient the δ function to be injective.

Bennett's reversible machine is based on the *state saving* technique, storing in memory any information that successive computations would erase. As someone may notice, such a machine may require a considerable amount of storage, so that Bennett's model is more a theoretical proof than a real implementation. To this extent, subsequent works on time and space requirements were conducted by Bennett himself in [20] and, later, by Buhrman et al. in 2001 [24]. Buhrman proved an upper bound to reversibly simulating irreversible computations. To quote from [24]:

> *Previous results seem to suggest that a reversible simulation is stuck with either quadratic space use or exponential time use. This impression turns out to be false: Here we prove a trade off between time and space which has the exponential time simulation and the quadratic space simulation as extremes and for the first time gives a range of simulations using simultaneously sub-exponential ($f(n)$ is sub-exponential if $f(n) = o(n)$) time and sub-quadratic space.*

Cellular Automata (CA) Cellular Automata is another example of an abstract model to support reversible computations. A CA is a discrete mathematical model based on the concept of a self-reproducing machine, used to simulate complex systems in several fields. It composes of an infinite number of identical finite automata called cells, which are placed and connected uniformly in space. Each cell

is supposed to be in one finite set of states such that the evolution of one cell also affects its neighbourhoods. More in details, a cellular automaton is reversible if, for every current configuration of the cellular automaton, there is precisely one past configuration. In the 40s, Von Neumann first conceived the concept of a self-reproducible machine that appears in [115]. Later, with his colleague Stanislaw Ulam, Von Neumann further describes a more sophisticated model of Cellular Automata in the book "Theory of Self-Reproducing Automata" [116], completed by Arthur Burks after his death. In the 1980s, Matthew Cook, the assistant researcher of Stephen Wolfram, also proved that this computing model was Turing-complete [33, 119].

Ballistic billiards model In the late 1970s, Ed Fredkin, Tommaso Toffoli, and other members at MIT envisioned the concept of ballistic computing. Instead of using electronic signals like a conventional computer, it relies on the motion of spherical billiard balls in a nearly frictionless environment. The state of the machine would proceed forwards along its trajectory, and under its own momentum, with only a small fraction of the signal energy being dissipated to heat upon each operation. The original physical picture Fredkin and Toffoli envisioned was that of idealized, perfectly elastic billiard balls on a frictionless pool table. Nevertheless, it was just inspirational, having no real implementation proposals. Later on in [52], the authors proposed a closely analogous, but more feasible, electronic implementation; the project involved charge packets bouncing around along inductive paths between capacitors. Even though idealized, the picture proposed by Fredkin and Toffoli became the starting point for the following research topics, such as the adiabatic logic and circuits.

Borrowing from this theoretical model, other researchers have investigated and devised reversible computational models. An example of that is proposed in [89], where the author develops a new algorithm to cope with the simulation of colliding particles in a reversible flavour. The primary goal of this work is to address the reversing of the collisions with no memory overhead, which typically requires a memory amount proportional to the number of collisions. Again, this results prove the power reversible computing promises with its new dimensions of reasoning.

Adiabatic circuits Adiabatic Logic is the term given to low-power electronic circuits that implement reversible logic. It comes from the notion of adiabatic process in physics, where given a closed system the total energy (or heat) remains constant. Current CMOS technology, albeit more energy efficient compared to similar technologies, dissipates a considerable amount of energy as heat, mostly when switching. Further, as the circuits get smaller and faster, the overall energy dissipation increases. To solve the energy waste due to transistors switching in CMOS, adi-

abatic circuits follows two fundamental rules: (*a*) avoid to *switch on* a transistor whether a voltage difference between the drain and source is non-zero; (*b*) avoid to *switch off* a transistor whether a current is flowing through. The first rule is rather straightforward to understand. Without going too depth in details of transistors' operating, we recall only that the total power consumption P_{total} is given by Equation (2.4).

$$P_{total} = P_{dynamic} + P_{shortcircuit} + P_{leakage} \tag{2.4}$$

Where $P_{dynamic} = C \cdot V_{dd}^2 \cdot f_{switching}$, and $P_{shortcircuit}$ $P_{leakage} << P_{dynamic}$. P_{total} is primary due to the first component of Equation (2.4), $P_{dynamic}$. It is related to the on/off switching operation of the transistor. When signals change their logic state in a CMOS transistor, energy is drawn from the power supply to charge up the load capacitance from 0 to V_{dd}. As a result, the dynamic power is proportional to the switching frequency $f_{switching}$, i.e. the processor's clock speed. The second component of power, namely short-circuit power, derives from finite non-zero rise and fall times of transistors, which causes a direct current path between the supply and ground. Nevertheless, this amount is usually negligible and not really significant in logic design. The third component of power dissipation in CMOS circuits is the static or leakage power. Even though a transistor is in a stable logic state, just because it is powered-on, it continues to leak small amounts of power at almost all junctions due to various effects. As for the second rule mentioned above, it requires some electronics details, which are out of the scope of this dissertation, hence we redirect interested readers to the literature on adiabatic logic [109].

Several designs of adiabatic CMOS circuits have been developed, during the time. From work proposed by Charles Seitz in [105], and further explored and improved by other groups [60, 77, 122, 49]. Some of the more interesting ones include Split-level Charge Recovery Logic (SCRL) [122], and Two Level Adiabatic Logic, or 2LAL, developed by Michael Frank [49]. Both of them rely heavily on the transmission gates, use trapezoidal waves as the clock signal. Other examples of interest are also adiabatic circuits built using current nano-materials such as silicon nano-wires or carbon nano-tubes since nano-electronics are expected to dissipate a significant amount of heat. Another example of adiabatic circuits which leverages SCRL is FlatTop, described in [11]. It is the first scalable and fully-adiabatic reconfigurable processor (FPGS), capable of efficiently simulating 2D and 3D reversible circuits of arbitrary complexity. FlatTop works by mimicking a stack of computational models sophisticated enough to render it just a proof of concept, rather than a real working implementation. By the same research group, noteworthy is Pendulum, a fully-reversible MIPS-style microprocessor developed by Carl Vieri in [113].

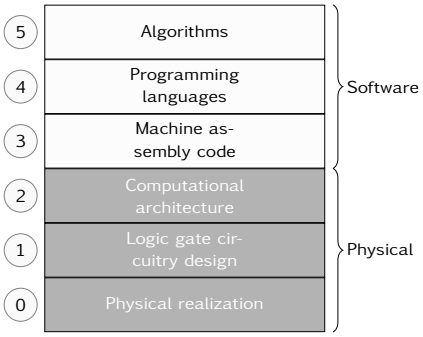


Figure 2.2. The layers stack of a reversible computer.

2.1.3 Software perspective

Alternative computational models and technologies for hardware architectures are, probably, the future of computer engineering. Though, they are still far from being a reality soon; notwithstanding the continuously increasing effort and money investments on looking for reversible hardware. In fact, computers still rely on conventional forward-deterministic logic, which impacts programming languages, algorithms and programs design. As human beings, it is equally true that our common way of thought is forward-deterministic, as well. It would require a great effort to train our minds to think "reversibly" and almost no one would envisage an algorithm in a reversible way from scratch. Harry Buhrman in [24] writes:

> *Currently, almost no algorithms and other programs are designed according to reversible principles [...] To write reversible programs by hand is unnatural and difficult. The natural way is to compile irreversible programs to reversible ones.*

Therefore, while the research on new hardware technologies proceeds, other researchers start addressing some challenges from a higher abstraction level. Sliding upward on the layers of the Figure 2.2 we approach software reversibility, which is the one we spent effort on, in this thesis.

The Definition 2.5 below gives the idea of what is the purpose of software reversible computing supports. The challenge of (general) software reversible computing is to allow the execution trajectory to switch back and forth without restrictions, by retracing the same computational steps. The goal is to approach a "perfect-reversible" solution that incurs (close to) zero CPU and memory overheads during both execution directions. However, a minimum quantity of overhead is required, due to the need for saving information that conventional computational architectures otherwise erase. In other words, we have to preserve information from the so-called *destructive* operations (see Table 2.1).

Definition 2.5 (*Reversible program*). A program is defined as reversible if and only if it is both *forward* and *backward* deterministic, at the same time.

From a software perspective, there are several application contexts where reversible computing can take place. In the following, we provide a brief introduction of the most relevant, introducing the one relative to our scope. Possible applications fields are: (*a*) fault tolerance (and fault detection), (*b*) asynchronous computing (synchronization problem), (*c*) speculative execution, and (*d*) debugging.

In its broader meaning, fault tolerance is the reliability-related property of a system to seamlessly overtake transient faults—definition of *faults* varies according to the application—and continuing the execution as if "nothing bad" had happened. Consistency and fault-tolerance are, thus, fundamental properties several complex systems must exhibit. On the other hand, synchronization is another key aspect, ensuring that the whole system obeys data consistency and logical dependencies. However, synchronization supports inevitably affect performance, since they introduce the overhead of orchestrating each working component. Part of this synchronization time is unavoidable and fundamental in nature, according to the application's design; nonetheless, what remains can be optimized. This optimization can translate into a tuning of the system's working parameters and in code design. However, we have to observe a trivial consideration: it is much simpler to devise systems which ensure a strict conservative execution order, at performance detriment, rather than conceive an elegant "parallelizable" solution that manually copes with component's synchronization. Tuning the system for the optimal overhead is somewhat tricky. It needs first to calculate a-priori a lower bound and, then, to achieve it in practice. We disclose some further details of this dissertation, the challenges mentioned above are the ones we will address with an innovative *autonomic* model of self-tuning (Chapter 4 "*The hybrid rollback strategy*").

The synchronization problem can be addressed by devising algorithms and techniques able to reorder the operations so as to find the better exploitation of the computing unit—a clear example is the one of speculative execution in hardware processors,—or by exploiting some non-blocking techniques, in other words by relaxing the logical dependencies. Relaxing data dependency constraints allows *transient* violations to occur. *Speculative execution* is a typical example, where the execution trajectory is assumed to be reversible, and working units perform actions optimistically even if they are not proven to be *safe*—the concept of safety is evaluated accounting for the application's logic (e.g. causality). Conversely to any conservative scheme, which avoids transient violations at all, speculative execution exploits a much higher degree of actual parallelism by leveraging subtle concurrent interactions among working components, otherwise ignored. Never-

theless, both in processor architecture—by employing an instruction pipeline support [111]—and in large-scale parallel (or distributed) computing, speculative execution has been proven to be an effective solution which reduces synchronization overheads and increases application's parallelism dynamically[2]. Each component executes asynchronously to each other, both in forward and in backward direction. Any misbehaviour is a-posteriori detected at runtime and fixed by undoing a portion of the forward execution trajectory; therefore we need some reversible support. At the same time, parallel applications try to harness as much parallelism as possible from the hardware underneath—consider the diffusion of multi- and many-core systems. This approach translates into an increased amount of concurrency due to computing units that have to interact asynchronously with each other, subject to generic failures which cause the system to lay into an incoherent state.

In the examples above, if a failure occurs, the rollback operation restores the system's state to a target checkpoint and re-establish its execution from that point over, wasting a portion of the computation though. Such a solution requires to maintain a program history. With reversible computing, we may realign the system's state through back-stepping, i.e. rewinding the forward computation until we reach a target configuration. Regarding latency, the trade-off depends on the number of instructions that have to be untied for memory saving. So far we have seen two approaches to build reversible support: (*a*) saving copies of variables to memory or (*b*) computing inverses of individual operations without relying on memory at all. The obvious and foremost plain solution is the former approach of *state saving* [65] (or its sparse state saving and incremental variants [117, 83, 15]). As the name suggests, the program's history is built by periodically taking a checkpoint during the natural execution. Although it is simpler to implement and requires low architectural complexity, it introduces a time overhead due to additional instructions in charge of saving values, and it may require an exceeding amount of memory to store old copies. Besides, the undo operation could potentially corrupt data locality in contexts whose different working components share the same memory partitions. Conversely, performing inverse computations allows to consistently reduce memory costs, as no additional storage latency is required and no extra processing time would be wasted for memory movement operations. Generally, main memory exhibits much higher latencies with respect to CPU's arithmetic units; therefore, it is much more efficient to perform inverse operations rather than relying on memory accesses—the so-called *memory wall* which refers to the gap between the speed of arithmetic/logic and memory movement operations. The goal is to overcome core challenges that limit the design of reversible applications, both for architectural complexity, concerning the end user, and for efficiency

[2]It is sufficient to think about eventual-consistency adopted by some major service providers, such as Amazon [114].

Operator	Statement	Meaning	Type	Inverse
=	a = b		Destructive	RESTORE(a)
+=	a += b	a = a + b	Constructive	a -= b
-=	a -= b	a = a - b	Constructive	a += b
*=	a *= b	a = a * b	Constructive	a /= b
/=	a /= b	a = a / b	Destructive	RESTORE(a)
%=	a %= b	a = a % b	Destructive	RESTORE(a)
«=	a «= b	a = a « b	Destructive	RESTORE(a)
»=	a »= b	a = a » b	Destructive	RESTORE(a)
|=	a |= b	a = a | b	Destructive	RESTORE(a)
&=	a &= b	a = a & b	Destructive	RESTORE(a)
++	a++, ++a	a = a + 1	Constructive	a--
--	a--, --a	a = a - 1	Constructive	a++

Table 2.1. Example of constructiveness property for most common operations.

aspects. Therefore, on the one hand, it aims to improve performance by leveraging on more clever usage of memory, on the other hand, it is steered to provide an autonomic and transparent framework. The latter would allow achieving the twofold objective of relieving programming models from overburden complexity and of further allowing for a seamless blending of different undo techniques to best suit application characteristics at runtime.

From a purely logical and theoretical point of view, every algorithm can be reversible-transformed provided that we have a sufficient amount of storage to keep track of its whole history [18]. However, this is not always feasible.

As an inheritance of this forward-determinism, our programming languages' statements can be categorized in two macroscopic families: (*a*) *constructive* and (*b*) *destructive* operations. The former ones are those in which no information is lost during the execution, whereas destructive operations are structured so as to overwrite one or more operands, making it impossible to revert them "out of the box". Table 2.1 shows a brief summary of the most common C operands along with their possible inverses. Due to the traditional programming languages constructs' nature, destructive operations are typically employed in any program. Integer division, shift and modulo operations, in general cause loss of information for arbitrary variable values. Integer division may cause truncation of the result; remainder after division has indeed no information about the dividend obtained, even if we know the value of the divisor. Analogously, both shift and modulo operations may result in bits getting shifted out or truncated, and thus getting lost.

Branches, assignments or other non-deterministic functions (e.g. random number generators, or I/O) are often destructive operations that implicitly gobble bits of information. Beside the entropy that those operations generate flipping the bits' state, they basically prevent to rebuild prior states by a backward retrace (recall

the example in Figure 2.1). For instance, an `if` statement is solved during forward execution basing on the condition's value. However the outcome of that comparison will be just thrown away after the execution takes one of the two branches. On the contrary, backtracking the `if` statement would require either to keep the information of which branch has been taken, or to save all the variables needed to rebuild the computation (i.e. the `if` condition). Let us consider the mock code in Listing 2.1, written in C.

```
1  int foo(int value, int offset) {
2      int value;
3
4      if (value < offset) {
5          value += offset;
6      } else {
7          value -= offset;
8      }
9
10     value *= 4;
11
12     return value;
13 }
```

Listing 2.1. Indeterimism of code branch implies irreversibility.

Reading the code in Listing 2.1 from line 1, it is not possible to know beforehand which of the two branches will reach the statement at line 10, at least until we do not evaluate of the `if` condition. Analogously, once the branch is taken, it is possible to backtrack this decision only if we re-evaluate the condition. However, in this case, we lost the information of what was the previous `value`; whether it was less than `offset` or, rather, the opposite. To solve this issue we could save the branch outcome in a separate variable `branch`, and use it to properly reverse `foo`. This scenario is shown in Listing 2.2, which depicts a reversible and semantically equivalent version of the previous code snippet. In this case the variable `branch` retains the output of the comparison, allowing to properly undo the operations, which are performed by the relative reversing function shown in Listing 2.3.

```
1  int foo_reversible(int value, int offset) {
2      int value;
3      static char branch; // Global variable
4
5      branch = value < offset;
6      if (branch) {
7          value += offset;
8      } else {
9          value -= offset;
10     }
11
12     value *= 4;
13
14     return value;
15 }
```

Listing 2.2. Reversible implementation of the code branch.

```
1  int reverse_foo(int value, int offset) {
2      int value;
3      static char branch; // Global variable
4
5      value /= 4;
6
7      if (branch) {
8          value -= offset;
9      } else {
10         value += offset;
11     }
12
13     return value;
14 }
```

Listing 2.3. Inverse function that undoes `foo`.

A different yet possible approach would be to make some inference starting from the outcome. Suppose to be at line 13 of Listing 2.1 and to have the result of *value*. In this case, we could infer that if `value` is negative, then the only possible case is that `value` $<$ `offset` was true. However, the reader may observe this solution implies that we have and we are able to understand the semantics of Listing 2.1; otherwise, the question would be «what operation should I execute?». This scenario introduces another possible approach to reversible computing. Indeed, several solutions have been proposed so far in the literature to address the reversibility problem at the software level. They span over several abstraction levels and approaches:

- Program state recording
- Source code transformation
- Code virtualization
- Reversible languages
- Instruction reversion

The example mentioned above requires *source code transformation* to create a new *reversible* code, provided its irreversible counterpart. Nevertheless, as it gets more sophisticated, this transformation entails code analysis to retrieve and understand the semantics behind the code itself—something similar for the assembly abstraction level is presented in [42], or at an even higher abstraction level in [38].

As previously introduced, the obvious and foremost plain software solution to build "rollbackable" applications is to *record the program state*, via the state saving [65, 66] technique. As the name suggests, the state of the computation is saved periodically or just before it would be modified by an operation. Therefore, as soon as the system is prompted, previous states are rebuilt by restoring the checkpoints recorded in the snapshot. Since they "sample" the program's state at discrete points in time, we need to choose the nearest checkpoint in time and replay those instructions which separates the checkpoint from the target we want to achieve. Although this solution is quite simple and requires low architectural complexity, it is highly time-consuming; moreover, it may still require a very large amount of storage, depending on the specific application. Optimizations have been proposed to improve the state saving strategy resulting in *sparse state saving* [16, 93] or *incremental state saving* [117, 98, 100]. With the latter optimization, only the directly modified portions of the state are stored, therefore it slightly lightens the memory requirements which, however, remains still considerable in complex systems, such as parallel and distributed applications. Further, unlike transactional applications, in general-purpose computation or in parallel simulation it is harder to recognize and keep track of variable modifications, making the design of the state saving routine more complex. Sparse state saving, instead, aims to reduce the frequency at which snapshots are taken.

Program animation is another approach to reversible execution [35]. It basically interposes a virtual machine with a reduced and reversible instruction set. Therefore, each real assembly machine instruction is uniquely mapped onto a reversible one allowing to run it backwards as needed. Though, the program must be dynamically interpreted and it slows considerably down the forward execution. In order to reduce runtime slowdown, a solution is to directly act at the source code level. Source code transformation is therefore targeted to statically parse an

irreversible code producing a reversible version of it by excluding destructing instructions ([102]); for those instruction, state saving is applied. Even though, time and memory requirements are lightened, they still slow down the forward normal execution.

Further on the different approaches to reversible computing, between 2002–2004 Tankut Akgul and his teamwork proposed an appealing solution towards the assembly *instruction reversion* [8, 7, 9]. They proposed a novel way to efficiently reverse the assembly code by generating a logically inverse copy which exactly unwinds the forward trajectory. It is an instruction-level reverse code generator which statically parses an input program and computes their reverse, instruction by instruction. Assembly level reversibility ensures a fine-grain rollback, without the need to forward rerun any statements. Stepping back to some previous instruction can be trivially done seamlessly, with regards to the user perspective. It simply diverts the natural control flow towards the inverse code, and back to the forward one as soon as the former point in time has been reached. Whenever the algorithm cannot straightway reverse the assembly code—as for the so called destructive operations which dispose some useful information,—it will fall back to state saving. Nevertheless, this kind of solution is also quite complex to realize and to implement. The major challenge of Akgul's solution is to infer software logic, whose semantic is mandatory to know how to group assembly instructions that belong to the same logical operation, hence how to reverse it. Akgul relies on dynamic control flow information from which he builds a Control Flow Graph (CFG) employed to properly reverse instruction blocks. Nevertheless, a burdensome effort is required to understand code instructions sequences and build the relative control flow graph. In his work, he exploits a three-passes static analysis to build a control flow graph in combination with a variable renaming algorithm which emulates the well-known Single Static Assignment form (SSA) [36].

Indeed, properly building an instruction-level reverse copy of the program introduces some other challenges. First of all, to be able to precisely infer the application's logic from a pure scan of machine instruction corpus. As already mentioned, programming languages and compilers are forward-deterministic, steered to easiness of use (by programmers) and machine code's optimization, respectively. Binary code is hence shuffled and tailored for the specific purpose. This requires to first unwind this elaboration in order to be able to recognize the logical execution trajectory within machine instructions. To this end, *reactive programming* [5, 6, 4] could be explored in conjunction with reverse computation, and more in general the data-flow analysis. The work proposed by Camil Demetrescu in [38] has proposed an extension to the C/C ++ languages where the programmer is able to mark a memory region (i.e., either complex data structures, or even single primitive

datatypes) as reactive. The executable, at compile time, is statically analysed in order to identify which operations can produce modifications on data.

Another field of research in the direction of reversibility hinges on devising *reversible programming languages*. However, the major hurdle is to write a reversible software in nature that overburdens the developer with a non-negligible extra effort; on the one hand it is practically difficult to design algorithms due to the non-conventional concealing and it is unreasonably time-consuming, on the other. Rather, it would be much more valuable to develop a system able to convert any irreversible program into a reversible one. Nonetheless, this "translation" requires a comprehensive knowledge of software and its semantic, especially if done automatically. Research done on high-level languages has built a solid theory, although mainly focused only on one-directional determinism structure, according to traditional machine architectures. On the contrary, backward deterministic languages' study is a quite new and thereby not much explored research area. Yokoyama et al., in [121, 120], face the principle that a programming language (independently of the abstraction level) must ensure so as to be reversible, further they devise a reversible high-level language prototype. Nonetheless, devising a program directly in a reversible flavour is not a straightforward process. This is why reversible languages, although feasible, are not practically employed. It would be much more convenient, in this case, to circumvent the problem by using a proper interpreter which generates the relative forward and reverse versions on behalf of the programmer, instead.

Applications of reversible execution through speculation at hardware level have also been proposed in 1993 by Herlihy and Moss [61]. They present a clever way to implement transactional memory in hardware—without relying on locks, at all—by modifying standard processor cache coherency protocols. The concept is quite straightforward and currently implemented by IBM and Intel in modern CPU architectures: any critical section is surrounded by a couple of instructions, namely `xbegin` and `xend`, and any attempt to access memory in critical sections are buffered into a private memory region. When the transaction ends, the system checks whether no conflicts have been detected and commits, otherwise the whole transaction is wiped out. Examples of reversible computing supports exist also at firmware level since several years. Pipelining in processing architectures is an example of reversibility applied at machine level. Processors' pipelining adopts a speculative scheme, e.g. in branch predictions. Instructions are executed optimistically until a consistency violation is met. In such a case, the entire work done is flushed and a new trajectory is re-executed from the misprediction point on.

As we will largely discuss in the further Chapter 3 "*The reversing framework*", we settle at the binary representation level. Before proceeding in exploring the solutions we propose in Chapter 3, it is worth providing also a more contextu-

alised overview on the speculative parallel discrete event simulation realm's related work.

2.2 Reversible computation in speculative PDES

The rollback operation is a fundamental building block to support the correct execution under speculative synchronization. In the context of Parallel Discrete Event Simulation (PDES) [55], schemes for undo-able execution have been widely adopted to address synchronization challenges [65, 45]. The Time Warp protocol proposed by David Jefferson in [65] proves the potential of rollback-based synchronization applied to discrete event applications (e.g. discrete event simulation)[3]. This protocol is built on a relaxation of the causal constraints, allowing execution to be logically reversible: operations are executed optimistically without ensuring the safety of the causality chain; rollback occurs just after the system detects a violation. Time Warp is relatively independent (in terms of its run-time dynamics) of both the simulation model's lookahead and the communication latency for exchanging data across threads/processes involved in the simulation platform. All these peculiarities allow it to guarantee high performance even in systems that are not tightly coupled and/or possibly entail up to millions of processors [87, 71]. According to classical PDES, the simulation model is partitioned into distinct simulation objects, which are mapped to Logical Processes (LPs). The latter is in charge of handling the execution of impulsive events, which ultimately produce state updates (hence transitions) in the actual simulation model state. As mentioned, Time Warp allows any LP to consume simulation events speculatively, regardless of consistency checks. As a result, events are processed independently of their *safety* (or causal consistency); as soon as an event is a-posteriori detected to be violating causality, its effects on the simulation state are undone, via the rollback operation. Building effective rollback support to the simulation state is, therefore, a fundamental building block for an effective optimistic simulation platform.

In the literature, generic rollback operations have been thoroughly studied, and the approaches that have been proposed provide benefits in specific scenarios. All those solutions can be mainly grouped into two separate families, namely *checkpoint-based* [65] and *reverse computing-based* [26], depending on the algorithmic technique used.

With checkpoint-based rollback, the engine exploits the knowledge of where the simulation state is located in main memory to create a copy of the simulation state after the execution of one (or a group of) operations which have produced

[3]Theoretical analyses have proven performance gains mathematically for Time Warp systems [10, 75]

state updates. To this extent, different possibilities have been presented, all aiming at reducing the cost due to creating a state snapshot, both concerning memory and CPU usage. Among the various research lines, we find two different main approaches, which have been often combined. On the one hand, solutions to reduce the frequency according to which a snapshot is generated, the so-called *sparse* or *periodic state saving* [76, 17, 80, 99, 46, 108, 94]—with a focus on detecting which is the best-suited checkpoint interval to minimize unfruitful work (e.g., taking checkpoints which are never used for a rollback operation). On the other hand, we find solutions which try to reduce the amount of data copied into a state snapshot, ensuring anyhow that no meaningful piece of information is lost at any time—the so-called *incremental state saving* [73, 118, 84]. A mixture of these approaches has been proposed as well in [86], trying to modify at runtime the execution mode of the state saving operation, depending on the current execution dynamics, to capture different execution phases of the simulation models. At different scales, all these solutions suffer from the high cost associated with making a (logically) complete copy of the simulation state, which is either proportional to the size of the state (full state saving) or to the number of update operations related to the execution of one or more events (incremental state saving).

The idea of supporting the rollback operation in the context of Time Warp simulation systems by relying on reverse computation rather than on snapshot restoration dates back to 1999 [26] from Christopher Carothers. In this work, the automatic generation of the reverse code relies on a custom compiler, namely rcc. This reversible compiler generates two versions of the input simulation model, the first is a reversible-instrumented version of the model, whereas the second is the resulting reversing code. An example of a hand-crafted reverse simulation model for a specific context is proposed in [104]. The authors developed and presented a new reversible model for the radio signal propagation, from its original forward formulation, with particular attention to illustrating of the runtime complexity behind. Thus, a comparative study with the classical state saving approach for the same problem is also described. Another attempt to automatically generate reversible code can be found in [72], where control flow analysis is used to generate code which allows reconstructing the execution path taken in the forward code. Differently from the work we have carried out in this thesis, in [72] reverse code is generated at compile time. This approach has the drawback of preventing, for example, the possibility to rely on any number of third-party libraries. In a more recent work in [102, 96], the authors perform source-to-source transformation of C++ code based on the ROSE compiler infrastructure [97], intercepting all operations which modify memory and recording information about the performed updates in a data structure that is used to reverse the effects of memory updates. We will recall this contribution in Chapter 5 "*Dealing with shared libraries*". The

effectiveness of reverse computing applied to the Time Warp protocol is proposed also in [14]. The authors illustrate the scalability of this synchronization protocol even on top of different computing architecture, as the IBM Blue Gene Supercomputers, assessing the performance of the system over a variety of workloads and using up to 2^{16} cores.

In contrast to checkpoint-based solutions, the reverse computing-based rollback operation tries to cancel the non-negligible memory footprint produced by the state saving technique. This solution grounds on the availability of reverse copies of simulation events [26, 72], such that if the execution of a forward event e on a simulation state S produces a state transition $e(S) \rightarrow S'$, then the execution of the reverse event $\hat{e}$ associated with e on S' produces the inverse transition $r(S') \rightarrow S$. Overall, while the reverse computing approach can strongly reduce the impact of memory usage from which state saving may suffer, the execution cost of the rollback operation is directly proportional to the execution time of simulation events. This cost could become predominant in case of events with high granularity, and the rollback length is non-minimal. Again Carothers, in his work [25], presents a modular C-based Time Warp simulation system capable of integrating several techniques, from the Fujimoto GVT algorithm [57] to reverse computation as he previously presented in [26]. However, as we repeatedly mentioned, our approach to reversible computing is slightly different from the one in the literature. We primarily focus on the memory footprint of the program, rather than on the semantics of operations/instructions.

By mixing the different philosophies standing behind the above state recoverability techniques, in this writing, we will present a new approach (Chapter 3 "*The reversing framework*") which combines the conventional undo logs with software reversibility (Chapter 4 "*The hybrid rollback strategy*"). Notably, in our proposal, the data that are typically recorded by the undo-log systems are used to generate so-called *reverse code blocks*, or *undo code blocks*. Orthogonally to the approach mentioned above we likewise investigate how to address the challenge of dynamically reversing code instructions of shared objects (Chapter 5 "*Dealing with shared libraries*").

Chapter 3

The reversing framework

Wait a minute. Wait a minute Doc, uh, are you telling me you built a time machine... out of a DeLorean?
— Marty McFly, Back to the Future

As we are interested in reverting the work done by an application, or more in general by an algorithm, our approach to *software reversible computing* focuses on the output it produces on memory. We take inspiration from the naïve observation that it is possible to somehow describe the algorithm's evolution through a sequence of generic "state transitions"; they eventually produce some output which translates into memory updates.

Memory is indeed a founding part of the whole computing process. Without it, we lack the information to move from one state to the other. Each state is the result of some computational steps, and its final output will be the input for its successor, according to some logic (i.e. the transition function). A generic state S hence represents a development stage, a partial result toward the outcome. This observation intuitively tells us that the computation of a generic algorithm A is a sequence of state transitions producing a *sequence of memory updates*. Leading from this considerations, we can assert that A is somehow described by a specific sequence of memory updates; it is what we refer to as *memory footprint*, and that we can resemble as the evolution's history of A.

We mentioned in Section 2.1.3 "*Software perspective*" that from a theoretical point of view, reversible programs (see Definition 2.5) allow moving along their evolution trajectory back and forth by undoing and replaying each computational step, respectively. Nevertheless, we can settle the meaning behind terms "replay" and "undo" in different ways, according to the approach to reversible computing. In Section 2.2 "*Reversible computation in speculative PDES*", we introduced a brief description of the conventional form of reversible computing; each computational step has forward and inverse versions that are executed accordingly to move forward or backwards. In this text, we address reversible computing look-

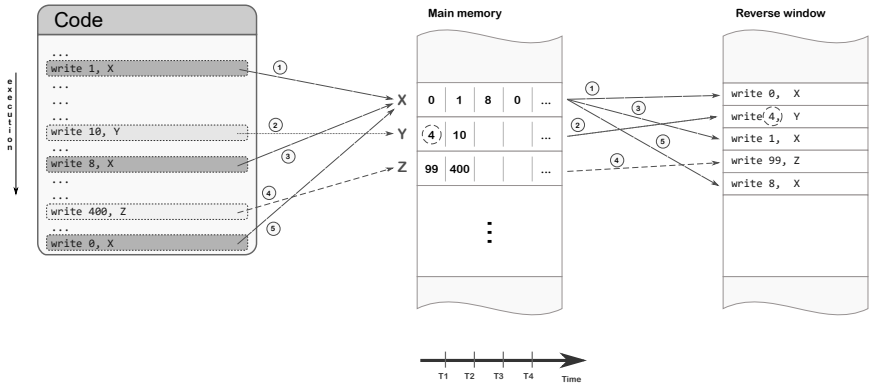


Figure 3.1. Memory footprint example of a program.

ing at the algorithms as a sort of "black-box", rather at the knowledge of their high-level semantics. Our definition of memory footprint does not carry any information about the application's semantics; it represents the "shadow" of what the application did, instead. Therefore, via this "footprint" we can re-bind the value of the significant variables so as to rebuild consistently the execution trajectory, even backwards. This flexibility is what we achieve by devising a framework capable of generating and keeping track of the memory's update history, so as to "reverse-transform" any generic application into a reversible one.

Indeed, things are a bit more complex, and this reversible-transformation comes at a price. On the one hand, it is possible to keep track of all the overwritten values, so that it would be possible to restore them as soon as required. On the other hand, however, it is not immediate to know at which memory locations to restore these values correctly, preserving the same application's behaviour. Further, we have to take into account I/O operations that, by their nature, are not deterministically reproducible; redoing the same I/O several times ends to different results on each run. These considerations pose some challenges. First, we have to devise a system able to handle information restoration efficiently and using as few meta-data as possible. Second, we have to consider a pseudo-deterministic behaviour in our reversed applications, and ensure a determinism level, though under specific conditions. The latter issue depends more on the needs and the objectives users want to achieve, which may need additional particular interventions. Vice versa, we solved the first issue by devising a structure, namely the *reverse window*, which merges the "executable" power of assembly instructions with their convenient means to convey data.

This whole chapter describes our reversing framework; how it handles the program image of a generic software application to make the program reversible in a transparent way, without the direct intervention of the user. Just as an anticipation, this result is achieved by relying on a technique to (statically) *instrument*

the program image (an insight about instrumentation is given in Section 3.2 "*Addressing code instrumentation*"). In our work's context, "making a program image reversible" basically means to enhance the code's capabilities, under the hood. As a matter of fact, we will keep a minimum yet sufficient set of information by intercepting a subset of code's instructions.

Section 3.1 "*Architectural details*" describes the reverse window's architecture and how it plugs itself within the applications' execution flow. Throughout the text, we should bear in mind that the reverse-transformation process of an application is carried out by a more complex framework (and toolchain) we will come back later at in this chapter, in Section 3.1.2 "*Reversing toolchain*". Section 3.2 "*Addressing code instrumentation*" illustrates in details the process of instrumentation, which is the strategy we rely on to modify generic code transparently to the user. Finally, in Section 3.3 "*Dealing with memory allocations*" we address the problem of the non-determinism introduced by "environmental" services providing memory allocation/deallocation facilities, such as the `malloc` library function. As a last note before moving on, we would like to recall that the development of the proposal in this work is done in C on a Linux environment.

3.1 Architectural details

Let us move forward and describe what a reverse window is[1], or by adopting our programmatic terminology what the `revwin` structure is. We already introduced the concept of memory footprint as a bare sequence of updates in memory. At the assembly level, these updates are made by instructions belonging to several families, e.g. `mov*`, `stos` and `cmov`, or in general by all those instructions having a memory location as the destination operand. Mentioned instructions modify the content of a target memory location with a new value at the expenses of the old one, which is *discarded*.

Our approach is to address the unavoidable necessity of saving erased data by packing it into a machine (assembly) instruction, rather than an accessory complex memory structure. The twofold advantage in this approach, beyond the visible means of data storage, is to harness instructions' power to embed also the "roll-back" logic in one fell swoop. By their nature, instructions convey sufficient information to known in advance how and where to replace data, relieving us of the cumbersome task of maintaining data consistency between the accessory structure and actual memory layout the application expects. In other words, instructions

[1] In this text, we will refer to the *reverse window* also as *undo code blocks* or *reverse cod blocks*. All the previous terms must be considered synonymous of reverse windows. This multiplicity comes from the fact that the last two terms subsume the real content of the reverse window itself, which is a mere set of instructions, indeed.

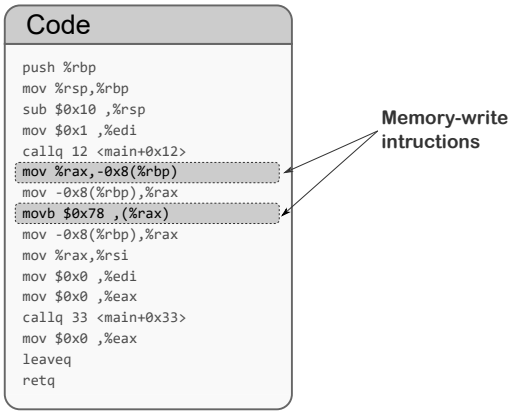


Figure 3.2. Example of a assembly program which makes updates on memory.

are a convenient means to do the right thing at the right place and moment, provided that we are able to properly generate them. This strategy allows us to reduce the overhead to computed and manage meta-data; they are needed to maintain the memory layout that (saved) values have to keep when further restored. As the reader may have noticed, it is quite straightforward that by creating a sufficiently detailed instruction-stepped map of these "inverse" instructions it would be possible to undo and replace even registers' value theoretically, though it lays slightly out the scope of our approach in this work. Nonetheless, it still constitutes a future perspective we will cope with—as mentioned in Section 2.1 "*Reversible computing*", interesting work in this direction is done by T. Akgul in [9]. On the contrary, what is relevant to our application scopes is the net data flow toward physical memory. It hence translates in focusing only on generic write instructions. Figure 3.2 gives a graphical clue of what it is happening.

The basic idea is to logically intercept every instruction responsible for propagating updates in memory, and save the value of the relative destination location just before the instruction replaces it with a new one. At this time we can proceed in crafting a complementary (inverse) instruction able to undo memory changes. From a high-level and theoretical point of view, it seems quite straightforward to achieve; how to do it practically, however? At this point, it enters the scene Hijacker[81][2], an instrumentation tool which allows handling code's capabilities of a generic program with some custom logic. We will address the instrumentation process further in this chapter in Section 3.2 "*Addressing code instrumentation*", however here we anticipate some details to give a clue about the technique employed to modify the executable code of the provided application. The instrumentation process has to ensure that is does not alter the outcome of the application, which we also refer to as the *original* or *native* application. Within the context of

[2]Hijacker is an open-source project; the source code is available at `https://github.com/HPDCS/hijacker`.

this work's effort, Hijacker is a building block we exploit to lay the foundation for the reversing framework this dissertation hinges upon. Nevertheless, before exploiting it, we have considerably enhanced its capabilities as it was in an early development stage. The choice of the tools can be found on its sketched structure whose flexibility promises significant customizability both for the matter of this writing and, moreover, to reuse it for future work orthogonally. Further on this chapter in Section 3.2 "*Addressing code instrumentation*" we will discuss more this tool, albeit its internals are not the scope of this text. The instrumentation process is fuelled by a binary file that represents our application's code. Through an XML configuration file, we instruct Hijacker how to alter the code of the input program. We remark that this modification is only a preliminary part of the whole reversing toolchain (see Section 3.1.2 "*Reversing toolchain*"). In particular, it will look for every instruction having side effects on memory and to inject, just before each of them, a brief preamble of additional code. Note that all this process lives at the machine code level. The injected preamble of code is in charge of triggering our *reversal engine*, namely the `reverse_code_generator()` function. As the name suggests, this function is responsible for creating at runtime a negative version of the original instruction from a logical, or better still, from the *memory* point of view.

Henceforth, we will adopt the following terminology: a *positive* instruction explicitly refers to the original one intercepted by the instrumentation tool, as opposed to its *negative* (or *inverse*) version dynamically generated by our reversal engine. The negative version of the instruction undoes memory updates by replacing the new value with the one discarded by the positive instruction.

3.1.1 The reverse window

We temporarily skip the details regarding our reverse toolchain, moving forward describing the architecture of a reverse window. It is the core structure of our reversing framework and its toolchain; it is devised as a C-structure that maintains negative instructions and potential data associated with them. Generally, the data is kept by the instruction's binary representation (instruction encoding); however, this is not true for string instruction families (e.g. `movs`) which require to act on a separate set of data. We will explore them more in details later in this chapter (refer to Section 3.1.3 "*The reverse code engine*").

It is worthy to remark that our reverse window is *atomic* by design. In other words, we envisage this structure so that it holds a set of instructions composing an "all-or-nothing" computational unit. In other words, a reverse window cannot be executed partially. This means that we are not allowed to perform a partial reversal of the memory updates the reverse windows have tracked. On the one

hand, this logic suites well the case of study we have undertaken, namely Discrete Event Simulation; on the other hand, it allows for a finer-grain control over the reversibility support.

To understand the structure of a reverse window and the reasons behind the implementation choices, we should introduce the scenario of a reverse window's execution. We recall that, in the remainder of this writing, we will refer to reverse windows also as *undo code blocks* (or *reverse code blocks*), subsuming its executable content conceived to revert the memory state.

Execution of a reverse window A reverse window contains a block of instructions that cancel the forward execution of some portion of computation. For the sake of simplicity and without loosing in generality, we will call this portion of computation an *event*. This notation happens to be convenient as it complies with the case of study where we will apply the results of this work and does not limit the general nature of the meaning. Now, as soon as the system is prompted to undo an event, the reverse window's code is invoked as if it were a conventional function. The reverse execution is requested passing through the `execute_reverse_code()` API function of our framework. This invocation entails the execution of all the inverse instructions contained in the code section of our reverse window—for the sake of brevity, we will say *execution of a reverse window* referring to this flow. The reverse window employs a pair `mov` instructions to restore the old value (see Figure 3.3). This choice is because of the x86 ISA standard does not provide a single machine instruction able to encode a whole quadword (64 bits) of immediate data. Thus, we must rely on an accessory register necessarily, and split the data movement into two steps: (*a*) first, the quadword representing the value to be restored is placed into the `rax` register[3]; once the `rax` register is set, (*b*) move the content of `rax` into the destination address already encoded in the inverse instruction (we will cover the aspect of how to generate it in the Section 3.1.3). Due to this technological impediment, we have to modify the initial content of the `rax` register on each reverse window's execution. However, we bear in mind that the execution of a reverse window is somehow transparent to the user application, which ignores its presence. Thus, just before the `ret` instruction returns the control to the original application, we need to realign the state of the `rax` register. A couple of `push/pop` instructions ensures consistency and performs this realignment.

Internal structure Now that we have seen the operating scenario of the reverse window, Figure 3.4 illustrates its internals. The C-structure is rather simple and

[3]The choice of the `rax` register is not arbitrary, rather it represents the only possible source operand for the subsequent 64-bits `movabs` instruction toward memory. We address to the Intel's manuals for mode details on that [64]

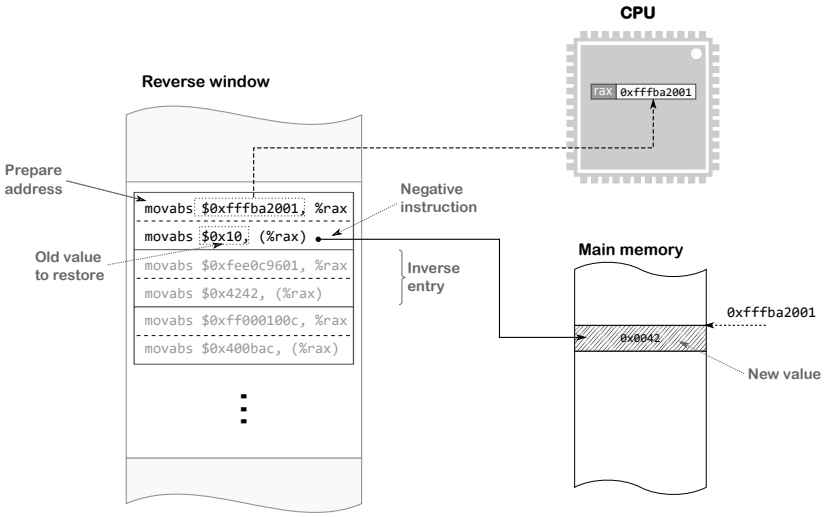


Figure 3.3. Illustration of the content of a single reverse window.

Field	Type	Description
`code_start`	`void *`	Holds the initial address where executable code starts.
`data_start`	`void *`	Holds the initial address where data dump starts.
`code`	`void *`	Holds the next free cell in the data dump where to continue to store.
`data`	`void *`	Holds the tip of executable reverse code, i.e. the entry point where to start undo execution.
`parent`	`struct _revwin_t *`	A link to the parent revwin, in case of overflow.
`raw`	`char[]`	Where the payload section of the reverse window starts.

Table 3.1. Description of the reverse window's field in C structure `struct _revwin_t`.

comprises a couple of pointers that Table 3.1 describes in detail. Among others, it is noteworthy to mention the `code` pointer, which points to the upper tip of the window and will be the entry point of the reverse execution; the `data` pointer to the area where to store chunks for `stos` or `movs`; and the `parent` variable holding a reference to another `revwin` structure in case of the reverse windows chaining—we will come back later on this point.

The reverse code generator feeds a reverse window, and crafts the negative instructions from each original memory-update one. The expansion order of the reverse window is the same of a stack, as a direct consequence of its nature to be reverse: the last positive instruction matches the generation of the first negative one, from which to start executing backwards. Leading from this naïve observation, we devised the reverse window so that code starts basically from the bottom

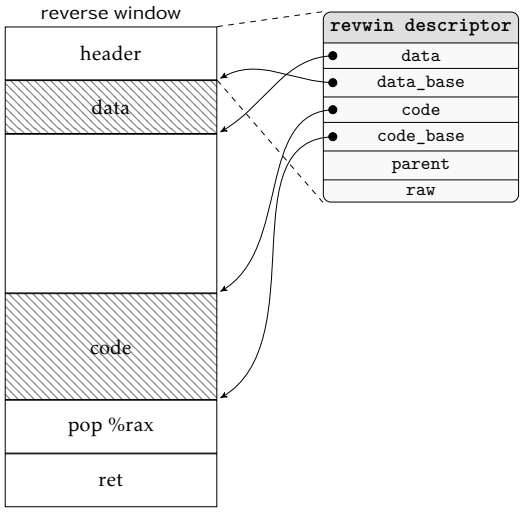


Figure 3.4. The figure illustrates the structure of a single reverse window, binding its descriptor's fields described in Table 3.1 to the actual representation in memory. Note that the reverse-window descriptor's `raw` field references a bare memory area where to store both code and data. At the same time, the window's descriptor is contained into the same memory area as an header.

of the window itself and grows upward; vice versa, the data section of the window grows from the top (just below the header) down to the bottom. This expansion continues until data and code sections do not overlap somewhere in the middle, possibly. The data section contains a collection of bytes the instruction is supposed to write in memory whose size, however, exceeds the instruction's binary representation to be encoded directly.

Each `revwin` structure is a pre-allocated slice of memory, handled by a simple internal slab allocator; hence the size of each window is fixed. At the creation of a new reverse window, the `revwin_alloc()` function is invoked. It reserves a free slab from the available pool and initializes it as a fresh reverse window as in the following. The initialization of a window places a `ret` instruction at the end of the window, and a `pop` instruction just before. This latter `pop` instruction will match a complementary `push` that will be placed as soon as the user requests the finalization of the reverse window by calling the `revwin_finalize()` API function. The presence of this pair of `push`/`pop` instructions, preserves CPU registers (i.e. `rax`) from our reverse window that will clobber them during its reverse execution, as mentioned above. Coming back to the initialization of a new reverse window, the `revwin_alloc()` function sets up `code_base` and `code` pointers; both of them hold the first free instruction's slot available in the reverse window. At runtime, the `code` pointer will increase each time a new inverse instruction is created and pushed into the window, whereas the `code_base` will always maintain the base address of the code section. In the same way, `data_base` and `data` point-

ers are handled to maintain the next available data slot and the starting address of the data section, respectively. The only difference is that the data section grows downward, contrarily to the code section that grows upward.

The finalization of a reverse window is performed by the above mentioned `execute_reverse_code()` function. This function will eventually trigger the execution of the reverse window itself, but before it finalizes the code section. In particular, it completes the reverse window by adding the complementary `push` instruction we discussed earlier; only after that, it launches the execution of the window's content beginning from the `code` entry point. From a practical point of view, the application will perceive the reverse window as a traditional function, invoked by the `execution_reverse_code()`, via a classical `call` statement.

Growth of the reverse window's content Each time the reverse code generator creates a new negative instruction (see Section 3.1.3 "*The reverse code engine*"), it is stored in the reverse window at `code` location. This pointer holds the next free slot in the code section of the `revwin` structure. The `revwin` structure is handled by our code generator which properly updates the metadata held by the structure in Table 3.1. Once the insertion is completed, indeed, the code generator automatically increments the `code` pointer of the negative instruction's size, so that it always points to the next free slot. If this is the case of a string instruction that needs to add additional data, the reverse code generator will use the `data` pointer; it then updates that pointer according to the size of the written data.

It could be the case that the free space to store new inverse instruction ends, where the `code` pointer reaches the `data` one and no more free space is available. In this case, an *overflow* occurs. Since the reverse window is built on top of a slab allocator for efficiency reasons, each window has a fixed size and cannot be reallocated. Once the window overflows, the architecture handles it by chaining a new "child" reverse window to the previous one, continuing to store instructions seamlessly. To chain together two (or even more) reverse windows, the `parent` pointer is used. This scenario is illustrated in Figure 3.5. The chain of multiple windows is handled by the framework both programmatically, through the `parent` pointer, and from a semantic perspective. At the end of the child `revwin`, the framework will replace the standard code epilogue, constituted by `ret` and `pop` instructions, with a direct `jmp` instruction to the parent window. The choice is compliant with the presence of a slab allocator and prevents to reallocate, e.g. through a `mremap` system-call or a `realloc`, the whole window. Both of the previous solutions would introduce an overhead due to the copy of bytes; rather, in our solution, we ensure the execution flows seamlessly with a negligible management overhead. Needless to say that a good early estimation of the window's size in the slab initialization may increase the overall efficiency of the whole reverse generation process.

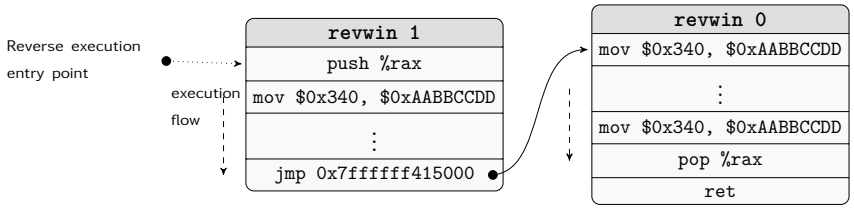


Figure 3.5. Chain of reverse windows in case of an overflow of the parent.

3.1.2 Reversing toolchain

So far, the previous section described the architecture of a single reverse window and its internals. Here, we address each step of the reversing toolchain; from the instrumentation phase up to the runtime generation of each negative instruction. The complete toolchain is described in Figure 3.6 and evolves by following these logical steps:

1. instrument the original application's code by injecting a *preamble code*;
2. prepare the stack for the subsequent invocation of the `reverse_code_generator();`
3. compute the destination address;
4. invoke the `reverse_code_generator()` function;
5. check whether the address has been already involved in a reverse instruction generation (we will come back later on this);
6. access the memory location (or area, e.g. for `movs/stos`) where the positive instruction will write new value V_{new};
7. generate and push a new inverse instruction;
8. return back control to the original instruction and continue the normal execution flow.

Looking at Figure 3.6, the whole reversing toolchain seems rather complex, though every steps adds a bit more of information needed by the subsequent one. The basic idea behind the modularization of the toolchain is to keep bounded the amount of additional code to inject, the runtime overhead and the complexity of the binary code itself.

Before actually reversing the memory-update instruction we have intercepted in the original code, we have to collect a few additional information concerning the instruction's encoding. The reversal engine must to know which kind of in-

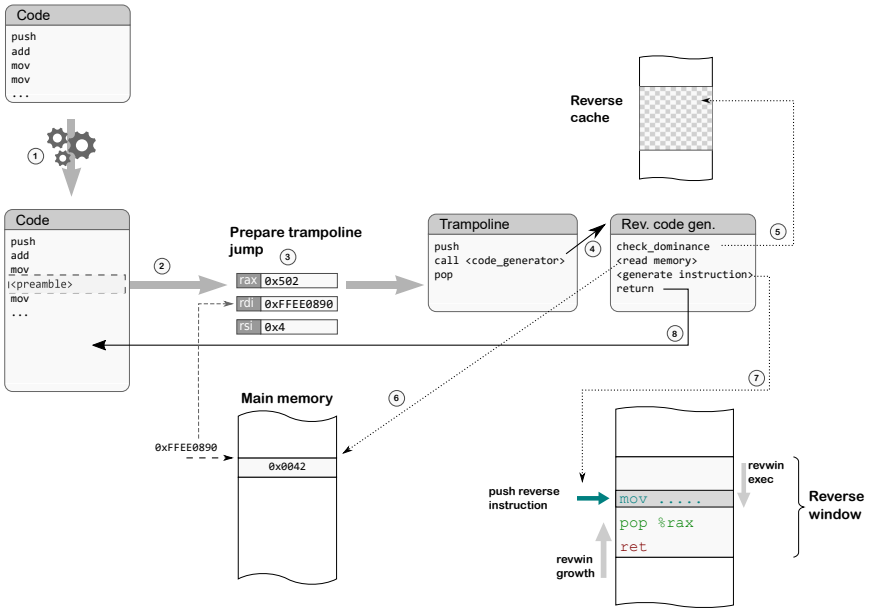


Figure 3.6. The reversing toolchain steps.

struction has to be reversed and where it points to in memory. From a practical point of view, the reversal engine needs the following basic information: (*a*) the size of the data that will be written, (*b*) the destination address where the instruction will write, and finally (*c*) some additional flags to properly handle conditional or string instructions. These data can be retrieved only by parsing the instruction binary representation they are encoded into; even worse, some of them need to be computed at runtime.

Passing data to the reverse code engine can be addressed basically in two ways. One solution is to use the stack by pushing data the runtime generator will eventually access. The second solution is to rely on an online instruction disassembler. This component would be in charge of decoding the target instruction at runtime to retrieve its flags and the relative destination address—we would stress that the instrumentation tool already has disclosed this information for its purposes. Disassembling a machine instruction is costly, as anyone can expect. Opting for the latter solution would translate into a waste of resources and performance penalty, since every instruction caught by our logic would require an invocation to the online disassembler. On the contrary, by relying on the results of the instrumentation phase, we harness the offline characteristic of the task which does not become an additional overhead for the runtime execution. We, therefore, embrace the former approach realized by two subsequent stages. This approach allows to minimize the amount of binary code injected per memory-write instruction and the overhead needed to prepare the runtime generator, at the same time. These two stages translate into as many code hooks. Each of them is responsible of building meaning-

ful information progressively, till invoking the final `reverse_code_generator()` function. Namely, we refer to them as the (*a*) preamble hook and the (*b*) trampoline hook. The combination of preamble and trampoline hooks, along with the execution of the reversing function `reverse_code_generator()`, represents what we call the *reversing toolchain*.

As we will see later on, we developed our instrumentation tool in such a way to decode and translate the instructions of the input application into a more convenient structure, namely the Intermediate Binary Representation (IBR). This step is the very first to integrate the reversing framework with generic programs. For the relevant instructions (i.e. memory-write instructions), the instrumentation tool wires part of the instruction's binary data into the specific C structure `insn_entry`, whose fields are described in Table 3.2). The `insn_entry` structure conveys the information needed by the reversal engine, and it is designed to be squashed into the instrumented version of the applications as a block of executable code. This portion of code constitutes the *instruction preamble* (or *preamble hook*) and represents the first preparatory stage before executing the trampoline hook. A first implementation of the preamble hook was a simple block of code that actually "pushes" the `insn_entry` (Figure 3.7) structure into the stack providing the information to the underneath trampoline. Noteworthy is that we adopt the x86 standard memory representation, which is evaluated from the tuple $\langle displacement, scale, index, base\rangle$ encoded binary representation; we will came back with more details on that in few paragraph. However, this implementation did not exploit some architecture's microcode facilities. We moved toward an optimized version (reported in Listing 3.1) which provides a speedup factor of 1.3X. Although the improvement may seem negligible, it must be considered in the whole context of several thousands of invocations due to each memory-write instruction encountered. It is important to note that this optimized version of the preamble code conveys the same information of the previous version, logically. As the only difference, it relies on registers, rather than on the memory stack, for data passing. For the sake of clarity, we show in Figure 3.7 and Table 3.2 the more comprehensive structure used by the instrumentation tool.

As the reader may have observed, in Figure 3.7 there is a `reversing function pointer` entry accounting for 8 bytes. The presence of this field derives from the general-purpose nature we aim for this framework. We implemented the instrumentation tool so that to be part of the toolchain of this reversing framework, yet bearing in mind to provide the user with a set of customizable features. This function pointer is given by the user during the instrumentation tool's configuration and could refer to any callback function the user wants to call when the instrumentation conditions are met. Since also these conditions are customizable by the

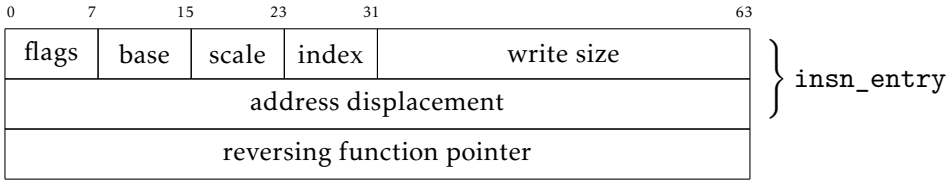


Figure 3.7. Instruction entry byte representation.

Field	Type	Description
`flags`	`char`	Instruction's flags, used by the instrumentation tool to identify `movs` instructions.
`base`	`char`	The base address register's identifier (if present).
`scale`	`char`	The scale factor (if present).
`index`	`char`	The index register's identifier (if present).
`size`	`int`	How many bytes the instruction will touch (if identifiable statically).
`offset`	`unsigned long long`	The executable entry point of the callback function.

Table 3.2. Instruction entry `insn_entry`'s fields

user, we have to guarantee the required flexibility, allowing to specify a custom callback function.

Let's come back to the preamble hook. At runtime, it propagates some information to the subsequent trampoline hook. Unfortunately, the destination address might be known only at runtime. Hence, even though we propagate the specific data associated with the target instruction through the preamble hook, we might need to emulate the machine's microcode behaviour to resolve it. Indeed, according to the x86 addressing mode the *effective address*—using the Intel's notation—is condensed into a tuple of data rather be plainly wired in the instruction's representation (Figure 3.8). As a result, memory addresses are associated with the tuple $\langle displacement, scale, index, base \rangle$, which is encoded in the instruction binary representation. The effective address is expressed as $base + (index \cdot scale) + displacement$. If the $displacement$ and the $scale$ are wired "as-is" into the instruction binary representation itself, parameters $index$ and $base$ are not plain values, rather they represent register's identifiers. The set of parameters $scale$, $index$ and $base$ is also known as the *SIB* byte, as they altogether account for 1 byte. To evaluate the expression, i.e. the destination address, we need to access the content of these registers; possible only at runtime. One of the main goals the preamble is one of resolving the *effective address* efficiently. Bearing in mind the x86 addressing mode, we designed our reversing toolchain architecture so to harness the `lea` machine instruction (line 8 of Listing 3.1). The `lea` instruction is fuelled with the same *SIB* byte and displacement of the target instruction for details about the in-

struction encoding). This solution leverages the underneath microcode, instead of spending precious clock cycles to emulates its behaviour.

Having a more in-depth look at the code listed in Listing 3.1, the more relevant parts are highlighted in a dashed box. We mentioned the `insn_entry` is "squashed" into a binary code; lines from 6 to 8 are the result of translating the C structure's contents into an optimized executable code. These lines are in charge of preparing the stack frame for the subsequent call to the trampoline hook. Respectively, they translate the following information for the upper code generator level: (*a*) the instruction's flags, (*b*) the number of bytes the instruction is attempting to write on memory, (*c*) the destination address. Instruction's flags are employed by the reverse code generation to check whether the machine instruction belongs to a specific class, such as string (e.g. `stos`) or conditional instructions (e.g. `cmov`). This check is preparatory to interpreting the addressing used and, in case of conditional instructions, to rebuild a consistent state.

The reader could ask himself/herself why we employed a `call` instruction instead of a simple direct `jmp`. Because the trampoline hook is a single binary code with the same entry point, it is unaware of its callee. Unlike a bare jump instruction, the conventional `call` guarantees that the microcode will properly handle the return flow, up to the callee native code.

```
1  pushfw
2  push %rax
3  push %rsi
4  push %rdi
5
6  mov <flags>, %rax
7  mov <size>,  %rsi
8  lea <sib>,   %rdi
9
10 call trampoline
11
12 pop %rdi
13 pop %rsi
14 pop %rax
15 popfw
```

Listing 3.1. The block of binary code injected as preamble before a memory-write instruction.

We reached stage 2 of the reversing toolchain, where the preamble code gives the control to the *trampoline hook*. This is an external module dynamically called before the actual invocation of the reverse code generator. Unlike the instruction preamble, the trampoline hook is not replicated for each instrumented instruction since it has a general purpose semantic and acts as the gate toward the actual re-

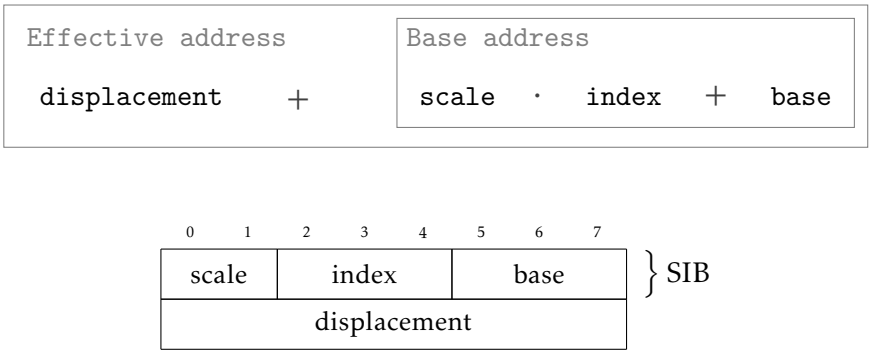


Figure 3.8. The x86's complete addressing mode. The SIB byte represents the tuple $\langle scale, index, base \rangle$, where parameters *index* and *base* are register's identifiers; we subsume their content, for the sake of brevity. Both *displacement* and *scale* are wired into the instruction encoding.

verse engine. The trampoline's code is relatively simple, having primarily the task of saving the CPU context so as to guarantee the native application a completely transparent invocation of the reverse code engine. For the sake of simplicity, we will not report here also the trampoline's code. The reverse transformation of the application takes place after the first compilation of the application's code; hence we have to explicitly protect the CPU context seen by the application by the impact of our revering toolchain. Compilers (e.g. gcc) generally can perform lots of instructions reordering to optimize different architectural aspects such as pipeline usage, cache misses, register spilling and similar; however, the compiler is aware of the whole code. In our case, on the contrary, the instrumentation process works on this optimized code and may breaks some assumption done by the compiler. Consequently, we have to manually save and restore the process image to seamlessly integrate our solution and to comply the calling conventions. Applications are unaware of the reverse code generation's behaviour which modifies registers' value during the whole process of inverse instructions creation, indeed.

Beyond the save and restoration of the CPU context, the trampoline code is also in charge of partially handling the case of string instructions, such as `movs` or `stos`. Because of their nature, this class of instructions requires a handful of steps more to evaluate the source operand, as we will see further in this section. The remaining part of the string instructions handling is performed by the `reverse_code_generator`, and so does also for other variants of write instructions, such as the conditional `cmov`.

3.1.3 The reverse code engine

So far, we presented how the reversing toolchain works and what a reverse window contains; but we have not seen yet how the process of generating a new negative instruction takes place.

Through instrumentation, the original program is augmented with additional binary code which works as a preparatory preamble to call the core function that actually crafts the negative instruction. The `reverse_code_generator()`, for the sake of brevity referred to as *engine*, takes three arguments:

`address`	the destination address where the positive instruction will write.
`size`	the actual size in bytes of the future write.
`flags`	the flags relative to the positive instruction to reverse.

Figure 3.10 illustrates the logic of `reverse_code_generator()`, which consists of the following tasks:

1. check whether `address` is already been referenced and dumped (cfr. Section 3.1.4 "*Instruction and object predominance*");
2. check whether to change the reversing strategy: (a) single or (b) clustered;
3. retrieve the value at memory location pointed to by `address`;
4. generate instruction's inverse;
5. store the newly created negative instruction into the reverse window (possibly with a dump of data).

After retrieving the parameters relative to the instruction, the engine performs a first preliminary check. Before generating the inverse instruction, it verifies whether `address` has been already referenced in a previous invocation. The reason behind this first step is to optimize the generation of ineffective inverse instructions. We will come back later in Section 3.1.4 "*Instruction and object predominance*" with more details on this, but for the moment it is sufficient to note that under certain conditions two references to the same address would be redundant. The address check employs a hash-based map we refer to as *reverse cache* (Section 3.1.5 "*Reverse cache*"). If it happens that the address has been previously referenced, there is no need to generate a negative instruction, under the instruction predominance property (Property 3.1), and the engine returns the control to the original code, right away. This optimization is done under the assumption that Property 3.1 holds; in this case, we can ignore the occurrence of this address.

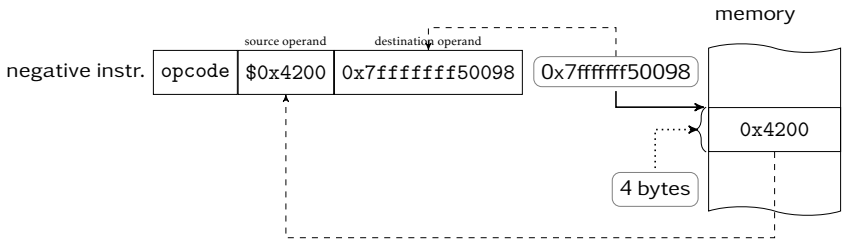


Figure 3.9. The engine creates an inverse instruction encoding in the instruction's binary representation the destination address as a destination operand and the current value of that memory location as the source operand. The amount of bytes to be read for this task depends on the argument of the engine's invocation.

Because a reverse window represents an atomic unit of execution, everything it contains is undone according to a "all-or-nothing" policy. Any address previously involved in the generation of a negative instruction is eventually dominated by the very first undo since it will be the last to be restored—recall that although the reverse window grows upward during its building, the execution flow is still downward. As an obvious result, it masks any previous undo which becomes useless.

On the contrary, if the address has not be referenced yet, a negative instruction must be generated according to the writing size. As for the crafting of the inverse instruction, the generator accesses the memory location pointed to by `address` and will retrieve the current value. This value is hence embedded straightway into the inverse instruction as the source operand (Figure 3.9). As the destination operand, the engine places the same memory address from which we read. It is interesting to note that the creation of a negative instruction is not affected by the logic in the forward mode, which can be of any arbitrary complexity. Indeed, since we focus only on the memory instruction's outcome, it is regardless of the computational complexity behind. In this way, we can guarantee an amortized constant time overhead needed to generate an inverse instruction.

The generation of reverse instructions is not a costly operation, except for the `movs` case where a memory buffer must be explicitly copied. Indeed, the set of instructions to be generated is very limited, and the *opcodes* are known beforehand. Therefore, we rely on a pre-compiled table of instructions in which only the memory address and the old immediate should be packed within.

Recall, now, that there are two classes of instructions which cannot be directly dealt with according to the aforementioned instrumentation scheme, rather they require special management; `cmov` and string instruction families. The former class is managed directly in the trampoline, which uses the `flags` field of the Figure 3.7. The 4 bits of this field (recall the Figure 3.7) records what is the check the trampoline have to emulate. If these bits are set, the corresponding status bits in

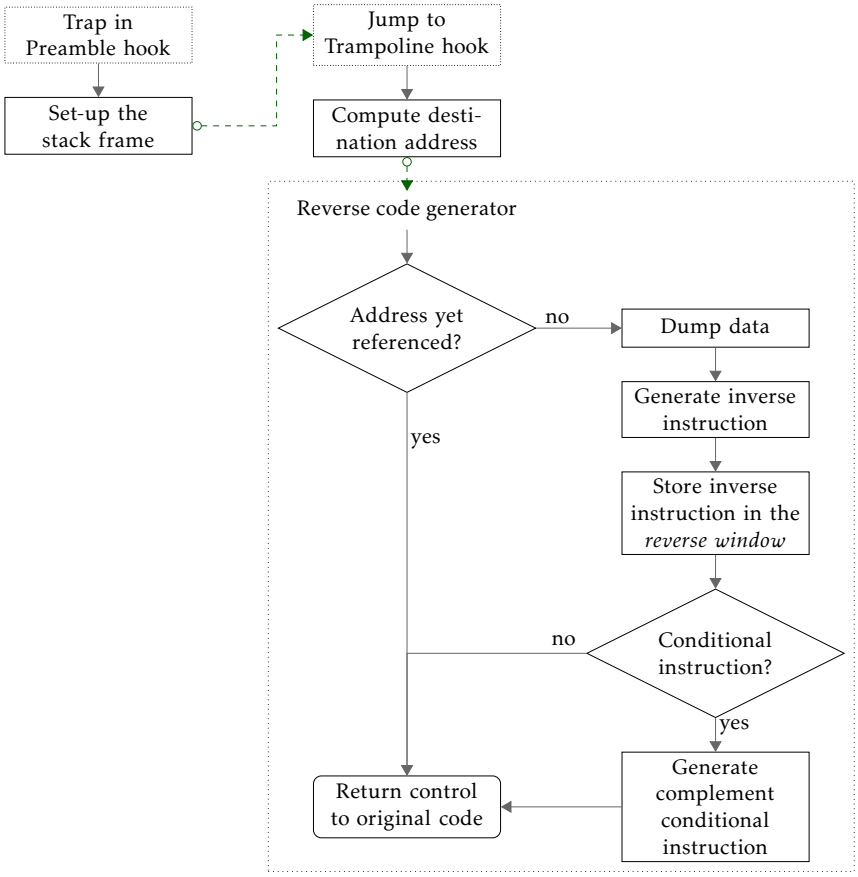


Figure 3.10. Flowchart of our reversal engine.

the EFLAGS register must be checked to determine whether the condition is met. Nevertheless, the values of status bits might have been already altered during the execution of the previous injected operations. To this end, the trampoline's code looks for the old value in the application stack, as stored during the CPU-context save phase. If the condition is met, the reversal engine treats `cmov` instructions exactly as the standard `mov`, otherwise the instruction is ignored.

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

3.1.4 Instruction and object predominance

It is noteworthy, now, to recall that our application context is related to Discrete Event Simulation (DES), where the smallest atomic unit of logical computation is an *event*. From a logical point of view, each event may convey several high-level operations; despite, it can be still perceived, under certain conditions, as a transaction which produces an *atomic* sequence of memory updates. Besides, an event may also cause several updates on the same memory location, as the effect of the application logic which progressively works on it (e.g. a do-while loop). However, from the perspective of the application's memory footprint, it translates into a mere sequence of several updates on the same regions. Each of this update should be traced and might be undone in case of reverse execution triggering. As the reader may have noticed, since in our application context the smallest computational unit is an atomic event that responds to the well-known "all-or-nothing" logic, it would be pointless to keep track of all the updates other than the first. Indeed, recall that our reverse window follows an inverse expansion direction; hence given a generic sequence of updates on the same memory location, former instructions dominate over the latter ones. From this simple observation, we derive the following Property 3.1 that we can harness in our application context as a feasible optimization due to the atomic nature of the event. As a side note, we would like to stress that the assumption of having an atomic unit of computation does not affect the generality of the solution in its whole. Indeed, it is always possible to identify a generic event as an arbitrarily small unit of computation, and tuning the

size of the reverse window to match any transactional logic; even in case of single machine instruction. This last consideration is where we start for the next future developments.

Property 3.1 (*Instruction dominance***).** Let's introduce a precedence relation $\prec$, which holds in a total ordered set of generic actions, and define the relation $\Diamond$ as the *dominance* operator. Let's assume also that the reversal operation's flow is not supposed to terminate on any of the instructions $i_k \mid \{i_k \succ i \wedge i_k \prec i'\}$. Any memory-update instruction i that touches a memory location $\hat{m}$ dominates another instruction i' if and only if instruction i updates, in the forward execution, $\hat{m}$ before i', namely $i \prec i'$.

$$i(\hat{m}) \Diamond i'(\hat{m}) \iff i \prec i'$$

To better explain Property 3.1, let us suppose to have a sequence of updates $A_1 \to B_1 \to C_1 \to B_2 \to A_2 \to D_1 \to A_3$, where each letter indicates a memory location where the update will take place and the subscript indicates a reference counter. We will call this sequence an *event* and Figure 3.11 will guide us. The relative reverse window generated will be clearly $A_3 \to D_1 \to A_2 \to B_2 \to C_1 \to B_1 \to A_1$. By proceeding backwards, we progressively reverse each instruction until we reach the target restoration point, that in our case is the event's beginning. Eventually, the last and durable restore operation is indeed the inverse of the first positive instruction that touches memory on that specific area. We stress that Property 3.1 is valid only under the assumption that reversal operations must logically comprise all the subsequent updates in a whole; in other words, if we are not interested in a partial reversal that could stop before what we consider the first update. It appears clear that keeping track of *all* the instructions updating the same memory address would be redundant, as it introduces an extra and useless overhead while undoing an event. To avoid unnecessary effort and costs, both for generating inverse instructions and the actual number of instruction to execute during the reversal operation, we employ a hashmap-based cache structure to keep track of addresses referenced during the forward execution of the application.

A further optimization aimed to reduce both the overhead in the forward execution, due to the generation of reverse instructions, and the overhead when executing an undo code block, due to the number of instructions contained, comes from the way software applications are often implemented. Structures and objects are commonly stored in the heap, for example, via `malloc` standard library's services or the `new` operator in object-oriented programming languages. We can extend Property 3.1 by defining the *Object Dominance* property (Property 3.2), where term *object* is used in a broader way with respect to the common usage in object-oriented programming languages. In our context, the term "object" identifies a

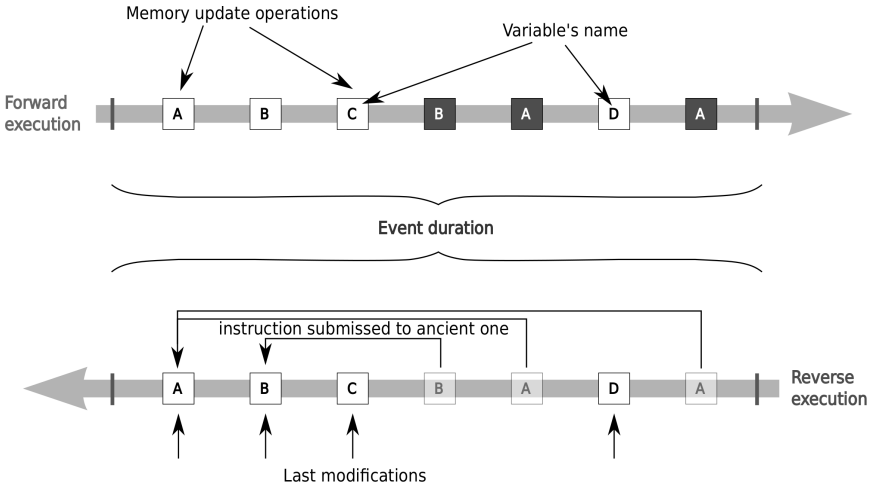


Figure 3.11. Illustration of the instruction predominance property. Every address that is previously involved in a negative instruction will be overwritten by the following negative instruction. In fact, the last reverse update on that address would be the one relative to the first reference to that address.

generic memory area keeping a portion of the application's state and instantiated by a single memory allocation operation (in the heap).

Property 3.2 (*Object Dominance*). Let i be an instruction executed by the event e in forward mode, and let o be an object spanning on the memory area M. If i accesses a single memory location $\hat{m}$ is said to dominate any other memory-update instruction i', executed later on by the same event e, if and only if i' updates a memory location $\hat{m}'$ such that $\hat{m}$ and $\hat{m}'$ belong to the same object o. Recall the operator $\prec$ as the precedence relation between two instructions:

$$i(\hat{m}) \lozenge i'(\hat{m}') \iff i \prec i' \wedge \hat{m}, \hat{m}' \in M$$

Property 3.2 is inspired to traditional time and spatial locality principles for software applications. It extends Property 3.1 laying on the consideration that if a structure's field or an object's attribute is updated during the forward execution, then other elements within the same (logically coupled) memory area will be likely updated as well. For execution patterns where Property 3.2 holds, it is much more efficient to generate simpler undo code blocks which restore a whole cluster rather than a single memory location. Restoring the image of the whole structure/object, as the application sees it at the beginning of the execution, results in a more compact reverse window and fewer negative instructions to generate and possibly executed. We refer to this technique as *chunk-based reversal*.

The reversal engine supports two operating modes that can be swapped dynamically at runtime according to the application's characteristics: (*a*) single-in-

struction and (*b*) chunk-based reversal. If the first is the default operating mode seen so far, chunk-based reversal is obtained creating a small snapshot of the whole structure/object before the first update operation that falls inside the memory boundaries of it is executed. The snapshot is kept within the reverse window in the *data* area (refer to 3.4), along with the instructions that are in charge of coping back the original content of the memory region[4]. The potential advantages of per-chunk reversal arise especially in contexts where memory update operations dominated according to Property 3.2 involve a non-minimal percentage α of the chunk area, which 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 4.3 of Chapter 4 "*The hybrid rollback strategy*", where we discuss a heuristic for determining whether it is convenient to switch to per-chunk undo code blocks or not.

3.1.5 Reverse cache

As mentioned, the reverse memory manager of our framework is equipped with a cache structure that keeps track of the addresses accessed during the forward execution of the application. This structure is defined by the `reverse_cache_t` C structure. It maintains a pointer to a linear array of words; each of them represents a line of our cache. The main purpose of this cache map is to keep track of what addresses are accessed and therefore which addresses are reversed within a single event. The map is held by the `reverse_cache_t` descriptor basically composed of a linear array of `reverse_cache_line_t`. Each element of the array represents a "line" of our reverse cache. Further details are provided by Table 3.3 and Figure 3.12 that depict the whole picture of the reverse cache graphically. Cache lines are queried directly by the reverse code generator to determine whether the destination address was already involved in a previous reverse instruction generation. The cache maintains also a counter of the number of subsequent accesses to each address and the usefulness of the cache itself. We came back on this aspect further in this section.

Each `reverse_cache_line_t` exploits a two-level bitmap approach allowing to coalesce multiple addresses within a single word to optimize the space requirement for address mapping. A toggle bit is sufficient to indicate if some memory-write instruction already references an address or not. The structure is a linear array of elements treated as a bi-dimensional matrix. Each element of the array is a quadword (i.e. 64 bits) used as the primary storage unit for a single range of family's addresses[5].

[4] Again, we harness the string instructions class already discussed.

[5] 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.

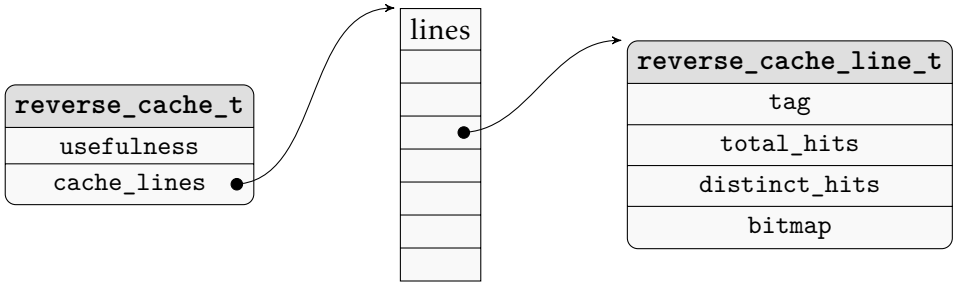


Figure 3.12. Reverse cache graphical description.

Field	Type	Description
`tag`	`unsigned long long`	Holds the identification *tag* of the cache line computed by masking the 26 more significant bits of the address.
`total_hits`	`unsigned int`	Holds the total number of touches to the same cache line, regardless of the specific destination address involved.
`distinct_hits`	`unsigned int`	Holds the number of touches concerning only *distinct* destination addresses.
`bitmap`	`unsigned int`	Holds an array of address ranges.

Table 3.3. Descriptor of a single line of our reverse cache structure `reverse_cache_t`.

To access the map, the following two values are needed: (*a*) an *index* providing the address family range, and (*b*) the *offset* which identifies the address' bit within the storage unit (i.e. the quadword). These two values are computed by properly masking the address. A *family range* comprises all the addresses whose value starts the same prefix. The length of this prefix depends on the number of flags the storage unit can contain. Namely a quadword, in our case, which can store up to 64 flags (2^n where $n = 7$). Given the address' value, the *offset* is computed by extracting the least $n - 1 = 6$ significant bits, while *index* is computed as the result of a bitwise AND with the remaining 58 most significant bits. Figure 3.13 shows an example of the address' binding for a 32-bit architecture—we do not report 64-bit case for simplicity.

3.2 Addressing code instrumentation

The main subject in previous sections was an enhanced version of the native application we want to reverse-transform, where a specific block of code is injected just before a generic memory-write instruction. This block of code belongs to the reversing toolchain and it is in charge of invoking the reversal engine at runtime. In this section, we do a step back in order to understand how this block of code is crafted and placed where it is. Addressing this issue, we laid on top *static binary*

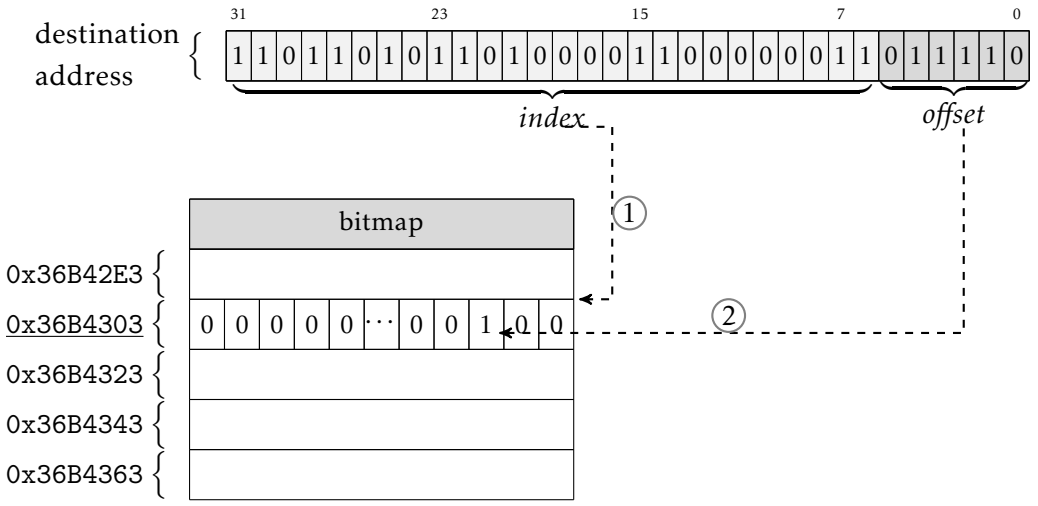


Figure 3.13. The `index` bitmask is used to first compute the family range (1) of the *destination address*; then we harness the `offset` bitmask to displace within the reverse cache to the related presence bit (2).

instrumentation to look for any memory-write instruction, transparently both to the original application and the user. Given the strong dependency with the underlying machine architecture, it is important to note that our work's context is a Linux environment on top of a x86 compliant instruction set architecture (ISA).

Instrumentation is the fine art of modifying blocks of code, or more generally a program, without affecting its outcome. Instrumentation may also refer to the act of monitoring, profiling or diagnosing a program's behaviour. Programmers quite often insert code chunks, directly or not, in their applications to have an insight at what is happening underneath; for example, to trace the execution path, to record values of internal variables, or to profile the program by counting the number of specific function invocations, for instance. Part of the instrumentation process is commonly handled throughout the design phase by the programmers themselves, who insert intelligible code snippets within any area they consider relevant, according to some criterion. However, this process may be rather intricate or exhaustive, for certain goals; instead there exist tools that address this process automatically. In this case, modifications have to be transparent to the application itself, which executes as if nothing were modified. Instrumentation process can be broadly classified into two main families: (*a*) static and (*b*) dynamic. The main difference between the two approaches resides on when instrumentation process takes place. Intuitively, the former is done *before* the application runs, either at compiling stage or at the source level. On the contrary, dynamic instrumentation occur directly at runtime *during* the program's execution. It means that the instrumentation process has to interpret the code to choose whether to trigger some

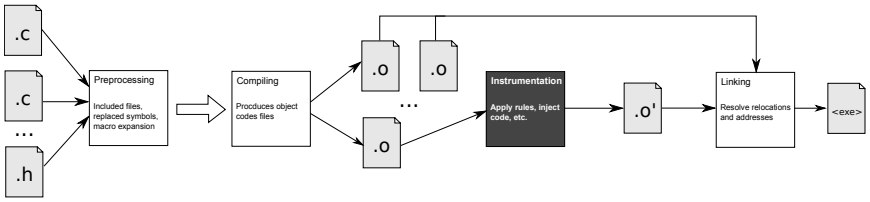


Figure 3.14. The Hijacker workflow within the compilation toolchain.

additional logic. Dynamic instrumentation, hence, adds a layer of abstraction between machine instructions and processors, somehow similar to what happens in a virtual machine.

Our static instrumentation approach, on the contrary, shifts as much as possible out of application's runtime the overhead of parsing, interpreting, and modifying the binary code. We settle just after the compiler's output and before the linker's one, in the compilation chain; hence at the machine code abstraction level. Binary instrumentation at this level, though, poses two significant challenges: (*a*) strong machine dependency and (*b*) user-side complexity due to the need of manually providing details on how to modify the existing code.

The advantage of working at the machine instruction level originates from the fine-grain information the binary structure provides, albeit it is not simple to treat or handle. However, overcoming this hurdle, the information provided by the binary structure can be employed for subtle optimizations of the reverse window, despite the overall reverse-generation procedure. This choice relieves from performing complex operations to understand the high-level logic behind programming languages, nor to infer in advance which information we have to store.

As the reader may observe in Figure 3.14, Hijacker is conceived to place itself within the compilation toolchain, specifically between the compiling and the linking stages. Hijacker takes in input a program image, which is first parsed and modelled through a high-level intermediate structure—to use the context's lexicon this structure is shortly called IBR - Intermediate Binary Representation. At this stage, the program image is not the final executable code yet; rather it is an object file containing machine instructions and their reciprocal references, which are partially evaluated into (relative) addresses. This kind of file is the so-called *relocatable* object file[6] that still needs linking phase to generate the final executable program.

The choice of relying on an intermediate binary representation is somehow obligated. It allows decoupling the machine architectural details of the target machine from the Hijacker's ones, as it creates an interface layer to speak with. This

[6]Our work targeted x86 architectures on Linux operating systems, therefore embracing the ELF representation for binary files

layer ensures the solution to be portable and flexible. Whereas, the choice to work on relocatable object files was mainly driven by the following two motivations: (*a*) the availability of low-level information which facilitates the construction of the internal representation by giving much more control on instructions, data and cross-references; and (*b*) because it much simplifies the emit phase, relieving it from the need to resolve relocation details explicitly. Linkable format[7], by definition, provides no semantic but conveys instruction-grained details (e.g. relocation placeholders) through which building the IBR is straightforward. The intermediate binary structure is used to interpret this "raw" relocatable object file and produce multiple (yet different) versions of the same program image, according to the configuration file provided by the user to the instrumentation tool. This technique, defined as "multi-coding". It is aimed to embed in the same final executable file different selectable behaviours of the same program that are not originally conceived to be together. We can choose which of these behaviours to adopted, according to the actual execution path and the relative objective we ought to pursue. Although working on the relocatable representation of executable files increases the user-side difficulties, relocations evaluation is still a responsibility of the standard linker. In such a way we do not only lighten the Hijacker's architecture, but we foremost ensure both modules' compliance, with regards to the environment, and software correctness, either syntactical and semantic.

Instrumentation tool Discussing the instrumentation process, we have to introduce the Hijacker tool necessarily. Addressing one of the core parts of this research, i.e. shaping the application's assembly code, we strive on a preexistent instrumentation tool, albeit in an early stage of development[8]. Hijacker was born at the HPDCS group of DIAG department of the University of Rome "La Sapienza". It is an open-source framework oriented at providing the necessary support for generic code alteration via static instrumentation at the assembly level. Nonetheless, we first significantly improved its capabilities and worked on a refactoring of its intermediate binary representation, which translates the assembly code into a C structure for more convenient handling. The major effort was to develop it as a modularized framework able to parse and emit the binary code, modified by a rule engine, fed by simple XML configuration file. A configuration is a set of rules that instructs the engine how and where manipulating the input program. In the future, a parallel effort of the group will strive for conceiving a set of "pre-cooked" configurations to be used "out of the box".

[7] In the whole dissertation, terms *linkable* and *relocatable* are used interchangeably, where not specified otherwise.

[8] Hijacker is available at the HPDCS website (`https://www.dis.uniroma1.it/hpdcs`).

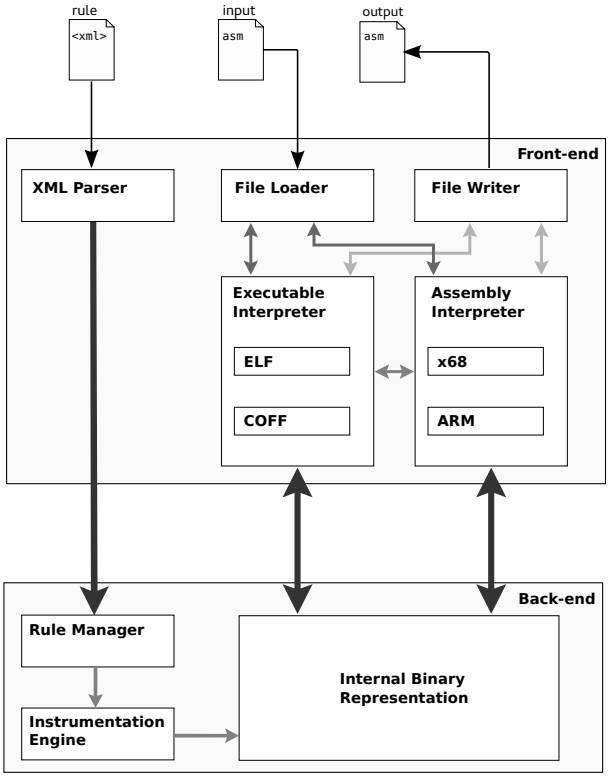


Figure 3.15. The Hijacker's architecture.

To give the reader a broader panorama of what is under the hood, it is deserving to spend some words about Hijacker's internals, which involves two main actors. Figure 3.15 shows the architectural schema of the Hijacker's internals. A front-end is responsible for handling the input and output relocatable object files. It interprets the underneath software's logic and the interactions among its parts and produces what we call the *Intermediate Binary Representation* also referred with acronym *IBR*. Through virtual references between descriptors, the IBR is a high-level convenient C structure. The file parser resolves symbols and relocation references between both instructions and data, and embeds the related information into the IBR descriptors; whereas the file writer converts back the IBR into a plain relocatable file. The IBR structure might evoke somehow the IR in LLVM or its equivalent for gcc, though it is much simpler and do not provide yet any interface APIs with who interact. Nonetheless, we employ the IBR interface structure to handle the binary files conveniently, concerning the programmatic level, in a similar way the mentioned IR do.

Besides, a back-end is in charge of instrumenting the program by means of its binary representation. It embraces both the rule manager and its counterpart, the instrumentation engine. The first read the XML configuration files and drives the latter altering the IBR accordingly. Whenever a rule is applied, the references

among instructions, functions and data are preserved since they are logical pointers. Harnessing the IBR, Hijacker can adds or remove instructions from the existing chain, regardless of the relations that other instructions have each other, which remains unaltered. Nevertheless, we need to perform a bit of adjusting on a subset of metadata. By far, the back-end is the module who introduces the most significant overhead, but considering that is a pre-processing submerse cost, we can consider it negligible. The instrumented application will not pays all this burdensome cost. Due to the extra instructions injected into the original executable code, anyhow, the instrumentation process introduces a slowdown at runtime too, which depends on the time spent along the diverted flow. It is not surprising, hence, that we devised the preamble code injected such as to minimize both the time and accessory memory usage, needed to accomplish the additional tasks of preparing the future invocation of the reversal engine.

Intermediate binary representation Section 3.2 guides us in describing the architecture of our Intermediate Binary Representation structure. A bare chain of instruction's descriptors represents the building block of our IBR structure. Each descriptor wraps a single machine instruction's binary encoding into a more convenient set of C data (Table 3.4). Providing just a high-level view on instruction chain, the IBR has also a list of function descriptors, where each descriptor points straight to the first and last instruction of the related function—we avoid to give more details on function descriptors as they are less relevant to have a clue of the instrumentation process. As the last ingredient, a list of symbol descriptors models what the ELF format identifies as relocations, which are a set of cross-references the relate other descriptors together. A single symbol descriptor (Table 3.5) defines the location, type, visibility and other traits of variables and functions declared, making possible to bind them also to raw data. Recall that ELF files are fundamentally a plain representation of bytes, therefore every reference between two pieces of code is expressed as a simple offset; likewise ELF format, it does the IBR.

3.3 Dealing with memory allocations

The last aspect to deal with in order to support a correct restoration of the computational state is related to the management of allocation/deallocation operations. Moving toward a general-purpose direction, in Chapter 5 "*Dealing with shared libraries*" we started to conceive an early memory management support for the reversing framework. In the mentioned case of study, we still rely on the memory manger of the simulation platform. Nevertheless, `libreverse` provides a module for memory management.

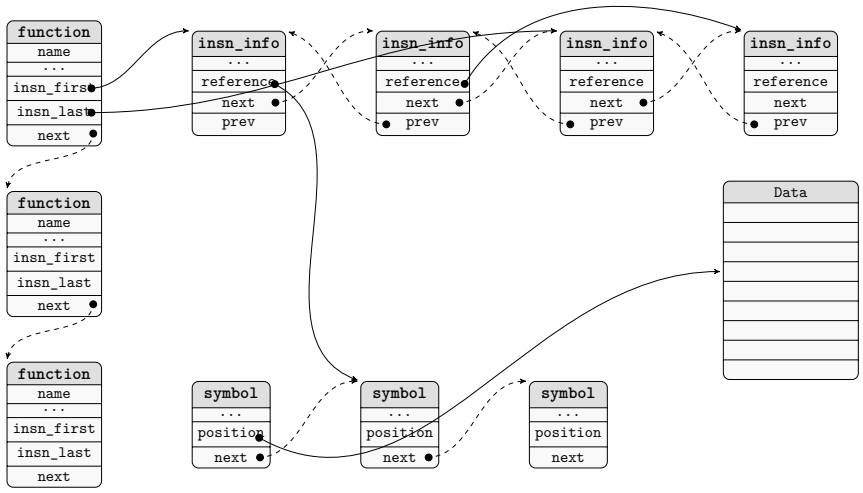


Figure 3.16. Intermediate binary representation

Field	Type	Description
`flags`	`unsigned long`	Keeps the *family flag* which the instruction belongs to.
`orig_addr`	`unsigned long long`	Keeps the address where the instruction is in the native code.
`new_addr`	`unsigned long long`	Keeps the address where the instruction is during the instrumentation process.
`size`	`unsigned int`	Keeps the size (in bytes) of the instruction.
`opcode_size`	`unsigned int`	Keeps the size (in bytes) of the whole instruction, except the possible immediate value or embedded offset.
`i`	`union::<info_insn_*>`	Embeds the machine-dependent descriptor of the instruction's binary encoding.
`reference`	`void *`	Keeps a pointer to the symbol descriptor referenced by the relocation (if any) during the instrumentation.
`prev`	`struct instruction`	Keeps the pointer to the previous instruction descriptor.
`next`	`struct instruction`	Keeps the pointer to the next instruction descriptor.

Table 3.4. The instruction descriptor's fields table

Field	Type	Description
`type`	`int`	Keeps the symbol's type in the Hijacker own specification.
`bind`	`int`	Keeps the bind's type in the Hijacker own specification.
`name`	`char *`	Keeps the pointer to the string buffer containing the symbol's name.
`size`	`unsigned int`	Keeps the size of one symbol entry in the relative relocatable file format.
`secnum`	`int`	Keeps the numerical ID of the original section to which the symbol belongs.
`index`	`int`	Keeps the numerical ID of the symbol itself within the original symbol section.
`position`	`long long`	Keeps the offset from the beginning of the section identified by `secnum`.
`offset`	`long`	Keeps the addend in the relative relocation entry (if any).
`reloc_type`	`int`	Keeps the type of the relative relocation entry (if any).
`duplicate`	`bool`	Keeps the internal flag used to determine if the symbol has duplicates; i.e. if there exists more references to it via relocation entries.
`referenced`	`bool`	Keeps the internal flag used by the parser to determine if the symbol has been resolved.
`extra_flags`	`long`	Keeps the info field of the ELF's symbol (either bind and type).

Table 3.5. The instruction descriptor fields table

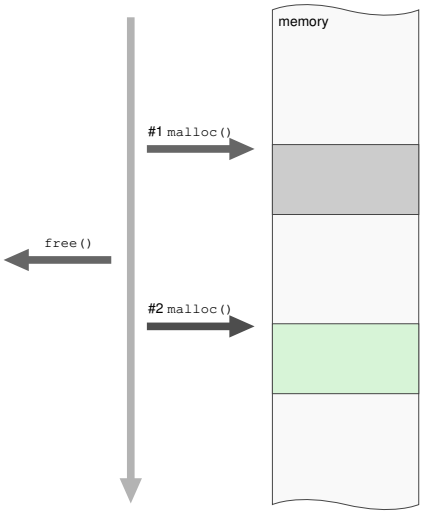


Figure 3.17. Subsequent invocations of `malloc` and `free` functions do not ensure the same memory area is reassigned deterministically. The figure illustrates the case of a sequence of `malloc-free-malloc` calls.

To better understand why this is needed, let's assume the scenario in Figure 3.17. As the reader can observe, the invocation of the standard library's function `malloc` is not piece-wise deterministic. If `malloc` is re-invoked after a call to the counterpart `free` was issued, nothing guarantees that the same memory area will be returned. To solve this problem, `libreverse` exploits a lazy policy in memory area deallocation. Through an internal structure, it keeps track of the deallocation requests but it does not really free buffers until the execution reaches a point beyond which the deallocation can be considered committed.

Additionally, the `libreverse` component provides a set of APIs to wrap external functions supposed to manage memory. Chapter 5 "*Dealing with shared libraries*" tackles in more details the challenges of a more general-purpose spectrum.

Chapter 4

The hybrid rollback strategy

Life can only be understood backwards; but it must be lived forwards.
— Søren Kierkegaard

So far, we walked through the foundations of this work; hereafter, on the contrary, this chapter—likewise Chapter 5 "*Dealing with shared libraries*"—discusses its main contributions, which leads from the work began in [29, 28, 31]. For the realization of this contribution, we rely on the open source architecture ROOT-Sim [85] developed by the HPDCS research group at the DIAG engineering department of Sapienza, University of Rome. ROOT-Sim is the reference simulation engine adopted throughout this text.

If Chapter 3 "*The reversing framework*" covered the process of crafting reverse code and its accessory structures, in this chapter we tackle how to exploit it in "real life". The contribution is not limited to a bare integration of our innovative reversible computing strategy in the context of parallel simulation. Rather, it goes beyond by proposing a new flavour of rollback support, which is realized as the combination of conventional state saving and our reverse window-based approaches. Leading from the famous Latin proverb "*in medio stat virtus*", we conceived the whole rollback support in such a way to leverage the mutual distance between the current time and the restoration point to decide from which direction it is more convenient to approach. Section 4.2 "*State restoration process*" largely describes our new *reverse scrubbing* rollback operation, in contrast to the *coasting forward* phase of the traditional state saving-based strategy. A mathematical model supports this decision system by enhancing the simulation environment with an autonomous behaviour. The resulting rollback support is able to tailor itself dynamically onto the workload of the application model, according to its variations.

The structure of the remainder of this chapter is as follows. We will first introduce in Section 4.3.1 the hybrid strategy and its integration in the Parallel Event Discrete Simulation realm, by a detailed digression on the implementation details. Therefore we will proceed in describing the mathematical model developed and

employed to this extent in Section 4.2, finally we will deserve Section 4.3 to a discussion on the experimental results of this model.

4.1 Hybrid strategy basics

In Section 2.2 we early introduced Discrete Event Simulation (DES) as a methodology to model real world's behaviours using a sequence of impulsive events that occur at certain discrete points in time. In this direction, Fujimoto thoroughly discusses in [55] Parallel Discrete Event Simulation (PDES), a set of techniques to exploit the parallelism achievable from the underneath hardware architectures. However, as the parallelism degree increases, the system needs a synchronization protocol that ensures to preserve the consistency among logical or causal dependencies. Time Warp [65] is one of the leading PDES synchronization protocol proposed in the literature. According to classical PDES, also in Time Warp the simulation model is partitioned into distinct simulation objects mapped onto Logical Processes (LPs). A single LP handles the execution of a sequence of discrete events; each event, in turn, ultimately produce updates to the actual simulation model's state, and may schedule other events to occur in the future. Given its speculative nature, also referred to as *optimism*, Time Warp leads the LPs to consume events independently of their causal consistency. Albeit there is a potential for great exploitation of the intrinsic model's parallelism, violations of dependency constraints can occur. Processing events out of the (virtual) time order, generates the majority of the causality violations, due to the arrival of *straggler* events—note that a violation, in the PDES paradigm, can occur within the realm of one LP. If some processed event is a posteriori detected to be violating causality, then a rollback operation must undo all its effects on the simulation state. A natural consequence is that performing the rollback operation, both correctly and efficiently, is a fundamental building block for an effective Time Warp system.

If a first consistent "preparatory" part of our research was to tackle how building reversing support transparently (see Chapter 3 "*The reversing framework*"), a second great endeavour has been facing both autonomicity and efficiency. Albeit the need for latter is more obvious to understand probably, autonomicity stems from the following observation. Simulation model's programmer—in general, target users are experts of a specific domain, not developers—should not care about synchronization implementation details at the programming language level, let alone at the machine instruction level. On the contrary, he/she focuses only on describing the real-world phenomena by means of mathematical models, which the simulation engine is in charge of executing efficiently. By casting ourselves in the context of this contribution, autonomicity and efficiency are ensured by the simulation engine which exposes a clever rollback hybrid strategy that mixes together

Strategy	Restore operation	Description
State saving	Coasting forward	Upon a consistency violation, the *coasting forward* operation re-aligns the execution state from the nearest previous checkpoint, according the restore time.
Undo code block (Reverse window)	Reverse scrubbing	Upon a consistency violation, the *reverse scrubbing* operation executes a sequence of binary code blocks (i.e. the reverse windows) that cancel the updates, perpetrated on memory by forward events, until the restore point is reached.

Table 4.1. Rollback strategies and their respective restore operations.

the two main families of rollback approaches: (*a*) state saving, and (*b*) based on reverse windows (i.e. the new reversible computing flavour we have introduced in this writing). Table 4.1 briefly summarize the associated restore operation employed by each of the aforementioned strategies.

Before going farther, it is worth to make a brief digression on some notations used in this writing. Hereafter, we will indicate with *forward execution* the traditional "top-down" execution flow of the application, in contrast to the *backward execution* that refers generically to the rollback operation from one state to a previous one. Note that the rollback operation can be, though, conveyed either by the traditional state saving or by the *reverse execution*. The latter term, in our case, indicates the specific process of executing an undo code block (i.e. the reverse window). Analogously, with *old value* we will refer to the value present in a given memory and that will be overwritten by a *new value*, resulting by the execution of a generic memory-update operation. Bearing in mind this introduction, we can now proceed in describing the realization of this hybrid rollback strategy.

4.2 State restoration process

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 is basically to undo the effects of e produced on memory during the forward execution, thus the reverse transition $r_e(S_i') \rightarrow S_i$. As we discussed in Chapter 3, reversing memory side effects associated with the transition $e(S_i) \rightarrow S_i'$ is the exact objective of the reverse window associated with the event e. We recall that the reverse window (or *undo code block*) is a set of machine instructions built at runtime that leverages the instrumentation done on the code itself at compile time, as we described in Section 3.1.1 "*The reverse window*". Each LP has a private memory area storing the relative portion of

the simulation state and has a private slab allocator for reverse windows as well. Given the nature of a single simulation event, only two outcomes are possible. Either the event is actually executed, and its effects on the simulation state take place *atomically*, or the updates must be wholly undone due to the transient erroneous execution of the event, i.e. a causality violation. Leading from this observation, we developed the system so that each event e has associated one reverse window ub_e that represents the event's "antimatter". Executing the reverse window annihilates the memory effects perpetrated by the forward execution trajectory of the event e. Since every memory update operation executed during the forward execution is intercepted by the instrumented code, the effect of the operation on memory is preserved by accessing the destination address location and saving its content just before the operation overwrites it. The sequence of the inverse of this operations composes the reverse window associated with the generic event e.

We emphasize that the reverse window is regardless of the operation/algorithm performed during the forward execution. Leading from the nature of our approach, it focuses on the memory footprint produced by the updates. The runtime generation process of negative instructions is, thus, unrelated to the knowledge of *how* the updates are made, but rather only on actually *what* is the outcome written to memory. As a result, the corresponding reverse execution of the undo code block does not entail recomputing the old value, let alone the knowledge of the algorithm that produced it. The reverse execution can be performed in constant time, regardless of the algorithm's complexity, anyhow it introduces a reduced overhead and keeps bounded the time wasted in the rollback phase, which is unfruitful in regards of the application's point of view. Nevertheless, an algorithm may update the same memory location multiple times, for example in case of algorithms using simulation state's variables as accumulators. Recalling the Property 3.1 in Chapter 3, we exploit it to reduce the number of negative instructions that must be generated, and therefore, the number of instructions be executed during the reverse execution of ub_e, as well. We also stress that this property holds under the assumption that a single discrete event is an *atomic unit of computation*. For a deeper reading of this aspect, we redirect to Section 3.1.4 "*Instruction and object predominance*".

A baseline scheme to exploit our solution is to recover the simulation state upon the detection of a causality violation, executing reverse windows associated with all the events to revert from the current live image of the LP's state. Figure 4.1 gives the graphical clue of this basic reverse scheme in rollback operation. The rollback chain, in this case, is represented by the events that separate the current simulation state where the causality violation (highlighted in dark grey) occurs from the restore time (in light grey). In order to make the state-restore latency independent of the rollback length, the exploitable approach is to rely on sparse

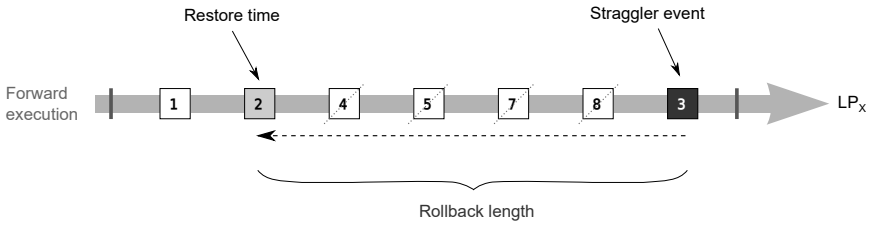


Figure 4.1. Basic reverse schema to employ reverse window in a rollback operation.

state saving (e.g. periodic checkpointing) *in combination* with our undo code block approach.

Let us recall how pure periodic state saving approaches achieve a restore latency independent of the rollback length—for a thourough discussion, refer to, e.g., [91]. Let us consider the case of a causal violation arising at time T_v associated with the event e_v. The state recovery procedure is, thus, based on identifying (*a*) the last causally consistent event $\hat{e}$, associated with the timestamp $T_{\hat{e}} < T_v$, and (*b*) one simulation state snapshot $\hat{S}$ such that $T_{\hat{S}} \leq T_{\hat{e}}$. However, once the checkpoint $\hat{S}$ is taken, an arbitrary number of events may be executed before the restore event $\hat{e}$; this is the typical case where $T_{\hat{S}} < T_{\hat{e}}$. Hence, reloading $\hat{S}$ is not sufficient to bring the simulation state back to the restore time $T_{\hat{e}}$, since the effect of one or multiple events in the interval $(T_{\hat{S}}, T_{\hat{e}}]$ would have been undone, as well, and erroneously lost. Instead, to properly realign the simulation state exactly to the time $T_{\hat{e}}$, the rollback operation must silently replay every intermediate events in the interval $(T_{\hat{S}}, T_{\hat{e}}]$, before returning the control to the normal execution flow. This specific replaying phase is the so-called *coasting forward* (recall Table 4.1). An illustration of the described recovery operation to simulation time $T_{\hat{e}}$ is reported in Figure 4.2a.

The cost associated with the coasting forward phase depends on two factors (as results, again, in [91]): (*a*) the average granularity δ_e of any event to be replayed, and (*b*) the actual number of replayed events. The event granularity δ_e is defined as the wall-clock time required to consume the event itself, producing its outcome on memory. Periodic checkpoint, is a variant of sparse state saving, as it relies on the basic idea to taking a simulation state checkpoint periodically after a certain number of events, rather than after every event. In such a context, a causality violation can occur at a generic point between two consecutive saved states. Denoting with χ the number of executed events between two consecutive snapshots S_1 and S_2, the coasting forward phase requires to reprocess $\frac{\chi-1}{2}$ events on the average, leading to a restore cost of $\delta_e \cdot \frac{\chi-1}{2}$. This makes the restore operation independent of the rollback length, but proportional to the distance between two consecutive checkpoints and the average event granularity δ_e.

(a) Rollback operation realized by the Coasting Forward operation.

(b) Rollback operation realized by our new *Reverse Scrubbing* operation.

Figure 4.2. According to the relative distance of the generic checkpoint C_x to the restoration point t_r, our hybrid rollback strategy combines the coasting forward operation employed by the conventional state saving strategy (a) and our new *reverse scrubbing* operation exploited by the approach based on reverse windows (b).

Embracing checkpoints, a complementary approach to coasting forward is what we call *reverse scrubbing*, employed when relying on undo code blocks. Again, periodic checkpointing make the state recovery operation independent of the rollback length[1]. Complementary to what coasting forward does, the reverse scrubbing operation reloads the checkpoint $\hat{S}$ associated with a timestamp $T_{\hat{S}} \geq T_{\hat{e}}$, from which the sequence of undo code blocks annihilates the effects of each event that separates chosen state snapshot $T_{\hat{S}}$ with the restore time $T_{\hat{e}}$. A corner case which might occur with this approach is that such a checkpoint does not exist yet since the simulation time is still approaching a new checkpoint period χ. In this scenario, we can simply promote the live state image of the LP as the starting scrubbing state checkpoint and run reverse windows from this point back. Figure 4.2b depicts an instance of reverse scrubbing operation where the restore point T_r, at virtual time 4, is two events before the fetched checkpoint, which coincides to the live state. The rollback algorithm reloads the state image and executes the reverse window associated with just one event (i.e. event at virtual time 8). Similar to the coasting forward case, the number of events to revert is $\frac{\chi-1}{2}$, where χ still represents the number of events between to successive checkpoints. Besides, the cost of the reverse scrubbing phase depends also on the average cost δ_{ub} related to the execution of a single undo code block, giving a total average cost of $\delta_{ub} \cdot \frac{\chi-1}{2}$.

Let us now consider the relation between δ_e and δ_{ub}. Undo code blocks are not full reverse event handlers since they revert side effects with no explicit recomputation of the values to be restored, so we may expect that especially for complex event processing logic, δ_e would be greater than δ_{ub}. As a result, via reverse scrubbing we may achieve, on average, the same state recovery cost with larger checkpoint periods χ, compared to coasting forward. Nonetheless, we have to take into account that reverse windows produces a forward computation overhead due to the instrumented additional instructions, absent in classical approaches purely based on periodic checkpoint. The overhead to generate undo code blocks at run-

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

time 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 discussed in Section 3.2 "*Addressing code instrumentation*", we achieve this via an optimized *multi-coding* scheme, still transparent to the application level software.

The combination of the forward and backward restore capabilities within a checkpoint interval is a topic that has already been investigated, particularly in [34]. However, it is 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 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 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:

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

δ_s	Average cost to *take* a snapshot.
δ_r	Average cost to *reload* a snapshot.
δ_{inst}	Average cost to *execute* instrumented events.
δ_{ub}	Average cost to *undo* event's effects.
ν	Average rollback length.
χ	Frequency at which to save system's state.
F_r	Average frequency of rollback.

In Equation (4.1), 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 [34]. 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 further on in Section 4.3 "*Experimental assessment*", 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 (4.1) 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.

4.3 Experimental assessment

4.3.1 Test-bed platform

In the matter of rollback support, we have fully integrated our reverse code block-based rollback support [30] within ROOT-Sim[2] [82], which we use as the reference PDES environment in this experimental study (and for the work in Chapter 5 "*Dealing with shared libraries*"). ROOT-Sim is a C-based open source simulation runtime environment, realized according to the ANSI-C standard and targeting POSIX systems, which implements a general-purpose simulation environment based on the Time Warp synchronization paradigm. It supports a fully-transparent checkpointing strategy (see [86]), based on the DyMeLoR memory map manager [110]. The same memory map manager has been exploited while integrating our reverse window-based approach to generate the combined (checkpoint/reversibility-based) restore technique presented in this chapter. In order to make the reader more comfortable with ROOT-Sim, Figure 4.3 gives a clue of the architecture of DyMeLoR, behind the simulation engine. As a last note, this simulation engine offers a very simple programming model relying on the classical notion of event's handlers. Each handler enables both for processing simulations events and for accessing a committed and globally-consistent state image, upon the GVT calculation.

4.3.2 Test-bed environment

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.

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

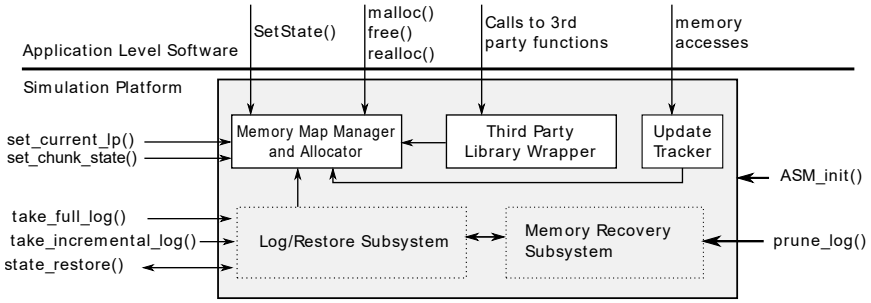


Figure 4.3. Architecture of the DyMeLoR memory manager of ROOT-Sim

Parameter	Value
RAM	64GB
Cores	32
Worker threads	32
Operating system	Linux Debian 6
Linux kernel	3.16.7-amd64
Architecture	64 bits

Table 4.2. Configuration of the environment for the test-bed application to assess the performance result of the hybrid strategy presented.

A summary of the environment configuration used to assess the strategy discussed until on is presented in Table 4.2.

4.3.3 Test-bed application

As a test-bed application, we chose a cellular system simulation model. The significance of this model is due to its complex simulation workload, which can be highly variable according to its configuration; for example, concerning memory the access pattern to the states of the LPs, and the intensity of the write activity on these states. Overall, this test-bed application allows us to assess our recoverability support by exploiting a suite of different workload profiles.

The application composes of many mobile communication cells modelled as a hexagonal coverage-area. Each LP handles the evolution of a single cell, by simulating the state of its hosted N high-fidelity radio channels. Channel model bears from the results provided in [68], where the authors describe how the interference and fading phenomena affect the experienced Signal-to-Interference Ratio (SIR). According to the results mentioned above, power regulation depends on the predetermined SIR levels to achieve, which is also explicitly modelled in this application test-bed.

The current state of any channel is kept via a dynamically-allocated record, placed into a list. As for the experiments, we consider two variants for the setup of the LP state layout; each one exhibits different tradeoffs. In the first variant, each channel's 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 as a whole block and are used to serve multiple call installations. Each block is made up of 100 records, and the different memory stocks are tracked by a hash-table, which is indexed by relying on the channel identifier. We will refer to the second variant as *clustered* (or *chunk-based*). Recall the Property 3.2 in Section 3.1.4 "*Instruction and object predominance*"; we can, hence, formulate 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. The information kept by the hashmap entry associated with a chunk is used to represent the so-called *chunk fragmentation*. This value expresses 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 chunk-based reverse windows. Switching to clustered-based strategy needs to consider the average fragmentation factor (among all the chunks) that can be easily computed as:

$$FR = \frac{\sum_1^n fr_i}{n} \tag{4.1}$$

The clustered variant allows improving memory locality since multiple power-management records belong to the same contiguous chunk. However, in this case, if a block uses a reduced percentage of entries the internal memory fragmentation, within the LP state, increases.

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 frequency of subsequent channel assignment requests to any target cell, according to an Erlang stochastic process [47]. 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 $\tau_{duration}/(\tau_A \cdot N)$. Due to the number of channel and power-management records to scan and update, increasing the utilization factors requires handling more power-management records at each time instant; eventually this scenario affects both the event granularity and the memory footprint on the LP state. The additional parameter $\tau_{handoff}$ 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

	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

Table 4.3. Summary of the test-bed configurations used.

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 an average value 120 seconds, while $\tau_{handoff}$ is exponentially distributed with an average value 300 seconds. To achieve two different values for the average wireless channels' utilization factor, namely 25% and 75%, we varied the mean value of the exponential distribution of the inter-arrival time τ_A. These two scenarios will be referred to as *Low-Load (LL)* and *High-Load (HL)*. The goal is to diversify the execution pattern of the application, both CPU and memory demand. Further, we consider two different read/write operation 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 (RI)*, 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 (RW)* profile.

On the basis of the varied parameters, a total number of six different configurations have been run, which are summarized in Table 4.3. 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 in the order of 80–100 μs for LL profiles, and in the order of 150–200 μs for HL profiles. The time to take a checkpoint has been observed to be in the order of 30 μs for LL profiles and of 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 in the order of 0.15 μs.

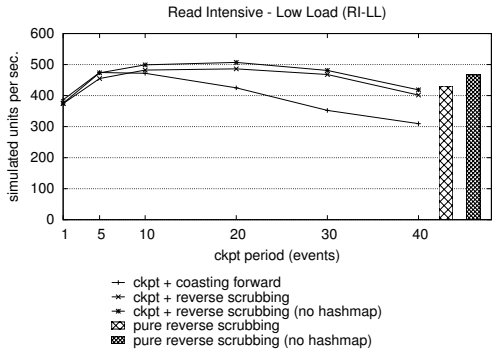
4.3.4 Performance data

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. 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 (4.1), will be assessed later via a second set of experiments.

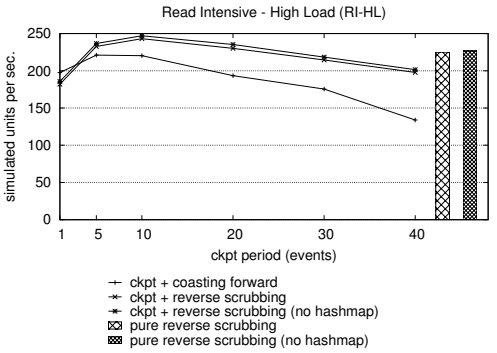
For completeness, we also consider a configuration where state recoverability is based on pure reverse scrubbing—no checkpoint is taken. All the configurations entail the generation of reverse windows, we run the experiments by either activating or disabling the hashmap to track already-updated memory 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 4.4 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[3]. The curves for reverse scrubbing, either in combination with checkpointing or not, refer to the case of baseline undo code block generation (see Section 3.1.3 "*The reverse code engine*"), 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

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



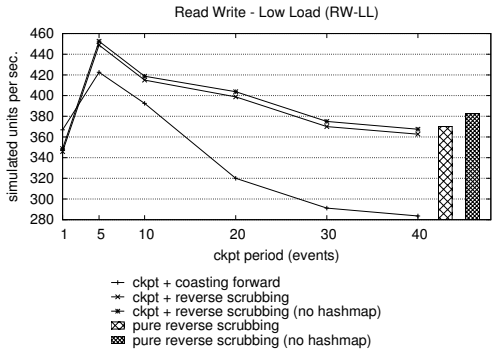
(a) Execution with a Low-Load workload profile.



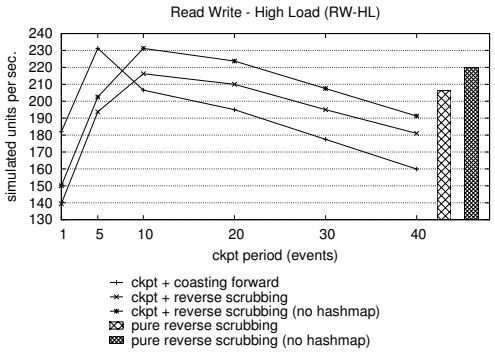
(b) Execution with a Low-Load workload profile.

Figure 4.4. Experimental results for the Read-Intensive profile.

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. 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 the generation of 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.



(a) Execution with a Low-Load workload profile.



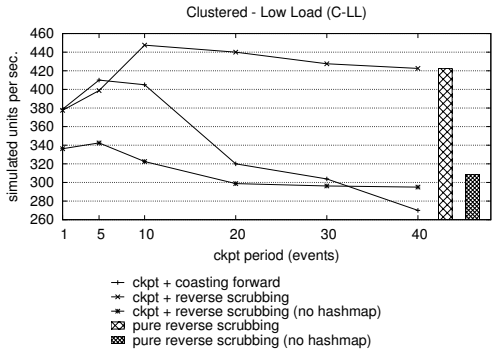
(b) Execution with a Low-Load workload profile.

Figure 4.5. Experimental results for the Read-Write profile.

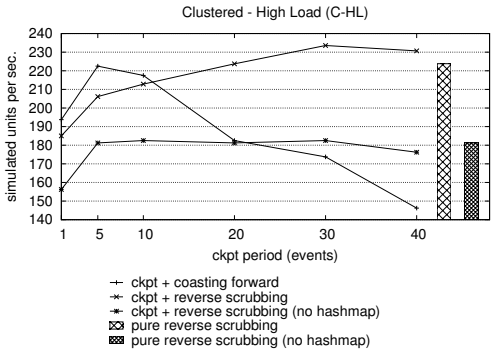
For the RI-HL configuration, the advantages by reverse scrubbing are amplified. In fact, HL gives rise to coarser-grain events which are adverse to coasting forward, especially for sub-optimal values of the checkpoint interval. In more detail, avoiding to take checkpoints at each event does not provide 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 leads to larger state/checkpint size, and more costly checkpoint reload. Finally, with intensive events which lead to query the hashmap infrequently given the RI profile of memory interactions, the overhead for managing the hashmap tends to disappear, as compared to the LL configuration.

In Figure 4.5, we show the results from the RW profile. In this scenario, reverse scrubbing still undoes memory updates via single memory-move instruc-

tions 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 hashmap. We remind that the RW profile still leads to memory updates mostly scattered on different locations, which prevents the hashmap from providing significant benefits. 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.



(a) Execution with a Low-Load workload profile.



(b) Execution with a Low-Load workload profile.

Figure 4.6. Experimental results for the Clustered configuration (RW profile).

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 4.6 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 pays off thanks to the statistical incidence of the reduction of the length of the reverse scrubbing phase (compared to pure reverse scrubbing).

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 this chapter. 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 (4.1). 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 (4.1) 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 (4.1) 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 [98].

The achieved results are plotted in Figure 4.7. 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 behaviour 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

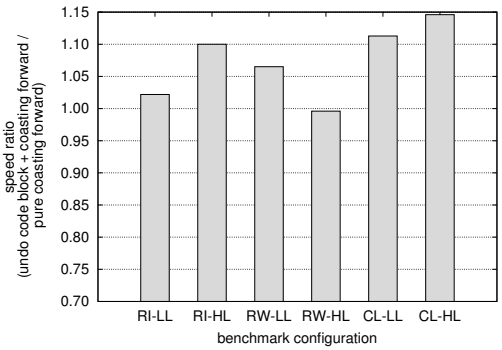


Figure 4.7. Execution speed ratio (new approach vs traditional periodic checkpointing).

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 favouring 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 4.5.

4.3.5 Comparison to state saving

The following additional performance data are related to the assessment of our hybrid recoverability technique, and its employment in combination with classical checkpointing. This study complements the one presented previously 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 (on-line) dimensioning/configuring of cloud oriented in-memory data platforms in face of specific data sets and differentiated data access patterns [39].

An illustration of the layout of the state of each LP, which models an individual server, is shown in Figure 4.8. 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.

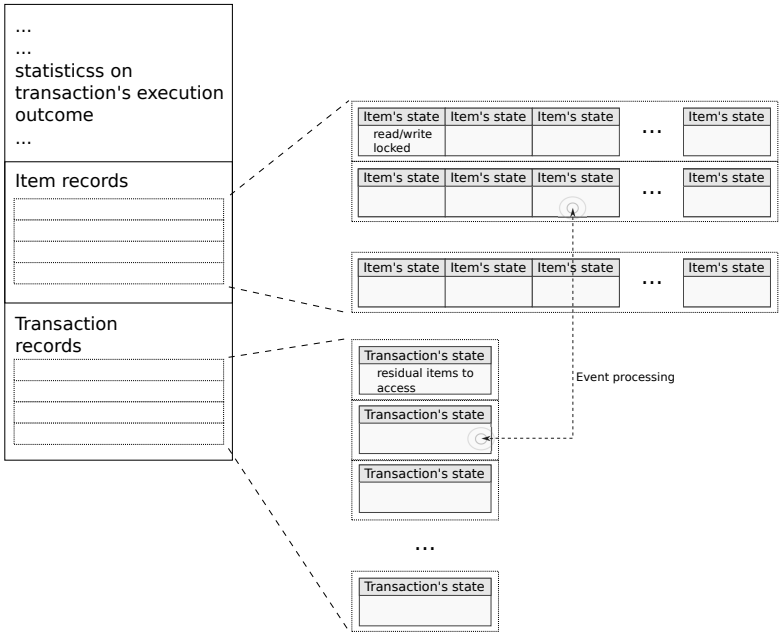


Figure 4.8. Layout of the LP state in the data store model.

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, from 5 to 10. Also, the evolution of the transactions in each batch is modelled by having their accesses processed in round-robin fashion. All the parallel runs have been carried out by relying on the same platform already described above, 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 earlier in Section 4.3.4, 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 μs while the latter is of the order 15 μs. 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.

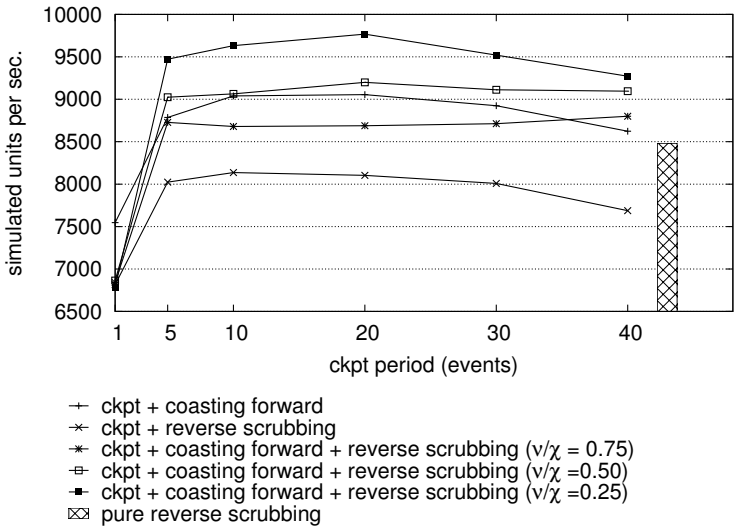


Figure 4.9. Results with the data platform model.

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 before. For the latter configuration we selected 3 different values of the ratio $\frac{\nu}{\chi}$: 0.75, 0.5 and 0.25.

The results are shown in Figure 4.9. 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 ($\frac{\nu}{\chi} = 0.5$). 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 ($\frac{\nu}{\chi} = 0.25$) 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 to non-minimal overhead even with fine grain events. Overall, in synergy with the results reported in the above exper-

iments described, 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.

Chapter 5

Dealing with shared libraries

Whenever a system becomes completely defined, some damn fool discovers something which either abolishes the system or expands it beyond recognition
— Finagle's Fifth Rule

The reverse computing-based rollback operation, which tries to cancel the non-negligible memory footprint of the state saving technique, relies on *reverse events*, which can be generated either manually [26] or automatically [71, 29, 102, 103]. With respect to automatic generation of reverse events, the various proposals address it by relying either on binary instrumentation [71, 29] or on source-to-source transformation [102, 103]. Nevertheless, none of these solutions is able to deal with third-party shared libraries, which could be regarded as an important building block for the development of complex simulation models, and applications in general. Just to mention some libraries, ALGLIB [107], GSL [59], FFTW [54], LAPACK [12], or BLAS [74] might be necessary for the description of statistical or algebraic processes, proper of a large number of simulation scenarios. Indeed, an endeavour in reversing the BLAS library is done by Kalyan Perumalla in [90]. The paper is focused on presenting performance and efficiency of two reverse versions of the library; a checkpoint-based solution vs. a computational-mode where restoration is obtained via inverse computation. The difference from the approach we propose here lies in the applicability of our solution, which we conceived to be general-purpose, despite we started settling it down of the specific context of the parallel simulation in this first proposal. It can be easily extended also to other contexts, since we tackled the roots of the problem on purpose.

Generally speaking third-party shared libraries are not *optimism-aware*. They are mostly devised for application scenarios which always operate on *stable* data; something that does never entail to be undone. This assumption is exactly the constraint speculative synchronization protocols, such as Time Warp, intentionally relax. Both the approaches of static binary instrumentation and source-to-

source transformation can make shared libraries *reversible*. However, as the reader may have noticed, their applicability might fail either (*a*) due to the lack of source code (in the case of closed-source libraries) or (*b*) because instrumenting shared objects produce system-wide effects. As a result of the last observation, with a very high probability, modifying shared libraries will also affect other programs not related to instrumentation purposes (e.g. optimistic simulation support). Non-speculative applications do not need exploiting this reversible-enabled version of shared libraries, which may introduce a non-necessary overhead, or even worse, could unpredictably hurt their behaviour. Further, for those speculative applications that demand reversible-enabled libraries, both the approaches would require to instrument the whole library, even though only a portion is really used, hence weakening the maintainability of the program.

We started from these observations to move a step further in the development of our reversing framework (Chapter 3 "*The reversing framework*"), nonetheless toward an orthogonal direction in regards to the contribution described in Section 4.3 "*Experimental assessment*" of this chapter. The need for treating correctly shared libraries brings us to discuss an alternative technique, which complements static and source-to-source instrumentation for x86 systems [102] and founds its roots on the concept of *lazy instrumentation*, resembling the lazy invocation of library's services. Indeed, our proposal specifically targets all scenarios where the source code is not available, that would prevent the source-to-source transformation from being a viable solution. The approach followed in this chapter borrows from the one we proposed in [31], integrating this novel instrumenting library manager into our reversing framework `libreverse`[1]. The core of our approach is a module that intercepts any call to third-party shared libraries' services and allows to create a reversible-instrumented version. Besides, `libreverse` provides also a set of management APIs that allow switching quickly between the instrumented and non-instrumented (i.e. original) version of library's services, opening to fine-grained runtime self-optimizations of the application run—it is similar, in principle, to the technique proposed in [86]. Looking at the simulation context, the possibility of choosing different behaviours of the simulation engine enables dealing platform code and simulation of the model separately. The former typically comprises housekeeping operations, so it does not require to embrace any reversible behaviour along a defined (critical) path where synchronization is not a concern; the latter, instead, is the scenario requiring to properly orchestrate distinct autonomous living entities, which generate and execute a flux of events. Overall, by relying on the approach proposed in this chapter, we enable speculative execution of any hierarchical combination of third-party libraries within a Time Warp-based simulation engine, significantly increasing the degree of programmability offered

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

to simulation models' developers. We stress again that this instrumentation process is transparent to the simulation model's developer, in addition, no change in the simulation model's code is required for the integration and a set of APIs allows tuning the reversible simulation engine at startup. Notwithstanding, the reader will notice that our approach is orthogonal to simulation realm; through our reversing framework, it is possible to extend this support to shared libraries in broader application contexts, beyond the specific one we used for the experimental assessment.

In [29, 30] we have used this support to generate "reverse events" as reverse windows (see Chapter 3 "*The reversing framework*"). The approach we describe here is orthogonal to the one mentioned in Chapter 3, where we do not address the presence of shared libraries. Shared objects, indeed, limit the use of static binary instrumentation at runtime, as every modification to their code would propagate system-wide. An approach similar in spirit to our proposal is that of Valgrind [79]. This framework gains control of the program's execution in a way similar to what we do and generate instrumented versions of the program code just-in-time (JIT). Nevertheless, since the goal of Valgrind is debugging, it relies on a virtualized CPU which runs the user program. This choice introduces a slowdown which can be affordable when debugging, but definitively not when executing high-performance simulations—more generally in any context where we cannot sacrifice performance. Conversely, we generate instrumented code which runs on physical cores, and interacts with the library manager only when needed, without relying on any virtualization. Our proposal is also related to many works in the field of program execution tracing (see, e.g., [1, 13, 92, 124]) for debugging, vulnerability assessment and repeatability. These approaches provide a detailed analysis of changes in the state of the program and the execution flow. Nevertheless, these works do not explicitly deal with the possibility to reverse a portion of the program's execution by relying on runtime-generated reverse instructions. The US patent in [62] explicitly deals with reversibility of shared libraries within executables. Conversely to our proposal, the goal is to make reversible the linking process, thus allowing for different versions of the library to be attached to the same program. Differently, we are interested in undoing the effects of shared libraries on the memory map and its content, to support a reversibility-based rollback operation.

The remained of this chapter first recalls, (Section 5.1), the basics of how the system dynamic linker supports dynamic symbol resolution so as to allow the reader to better understand the background behind our approach. Therefore, Section 5.1.1 and Section 5.1.2 discuss the proposal's internals and the methodology employed to implement the technique in a real-world scenario.

5.1 Resolve the symbol's address

Before discussing the approach that we undertake to enact software reversibility of generic third-party shared libraries, let us summarize how third-party libraries interact with an executable, taking as reference Linux systems relying on the Executable and Linkable Format (ELF). In complex systems in order to harness as much as possible the main memory, it has been introduced the concept of the virtual memory. Virtual memory is an abstraction provided by the Operating Systems to give user programs the illusion of having a large amount of continuously-addressed memory at their availability, which is achieved by mapping properly portions of code within the program's address space. Besides, quite often programs are built upon a plethora of modules that provide specialized and yet optimized services; namely the (shared) libraries. They are a comprehensive set of general-purpose facilities that developers can exploit as building blocks designing applications. To make the output of the compiler as much dry as possible, modern OS implement a sharing policy of these objects. Because of their nature, shared libraries require runtime address evaluation to properly route the code to the right managed entry point, according to the required facility. Albeit this task could appear rather straightforward, it is a burdensome task; in charge of which it is the dynamic linker `runtime_symbol_resolver()`. Note that, in this writing, we will use the term *dynamic linker* and *symbol resolver* to refer the same entity.

In standard Unix-like systems, the startup of a generic process begins with a call to the system-call `exec*` and passes through several nested calls to other preparatory functions until finally reach the real application's `main` function; Figure 5.1 depicts the whole picture. Without entering the complete details of this "journey"—it would lay outside our scopes—the steps relevant to our goals are the execution of `preinitarray` accessory functions and the `constructors` ones. We will used these facilities to "hijack" the system symbol resolver toward our custom resolver, by means of 2 injected functions, namely `_libreverse_preinit()` and `gotplt_hooking()`.

Before proceeding further and in order to better understand our approach, we need a digression to recall some basic notions proper of the ELF object files this solution heavily relies on. ELF file's structure funds on the basic notion of *symbols*, which is a rather generic term employed by the ELF format to describe different entities such as functions and variables. From a practical point of view, a symbol is just a pair $\langle address, offset \rangle$, which eventually translates into an *effective address* (adopting the Intel's lexicon). Hereafter, we refer to a symbol with its general meaning conveying the subtle notion of a memory location identified by the aforementioned tuple. Given that brief introduction on symbols, let's move on and unwind the dynamic symbol handling.

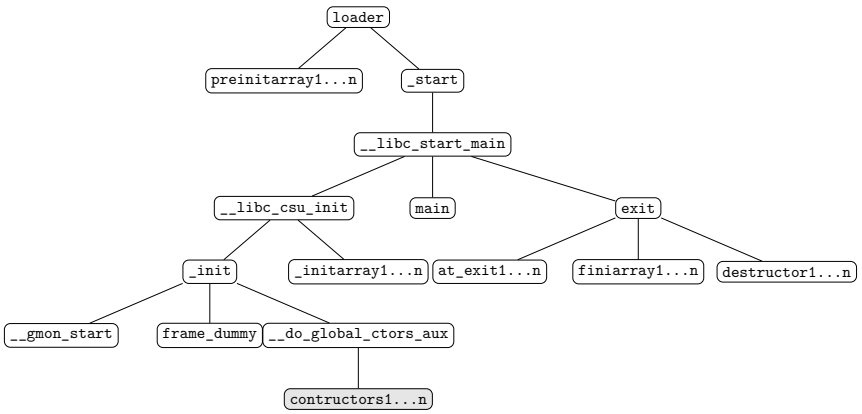


Figure 5.1. The whole picture of the functions call chain to start a new process in Unix-like Operating Systems.

Whenever the compiler determines that some function referenced in the source belongs to a shared library, it introduces in the ELF program's image additional pieces of information to let the system *resolve* any reference (at runtime) to that function towards its actual implementation. As for the resolution of shared symbols, the standard runtime environment relies on two accessory symbol tables that are wired into the ELF file, (*a*) the Global Offset Table (GOT), and (*b*) the Procedure Linkage Table (PLT). In particular, the compiler performs the following steps:

1. it records the name of the shared library in the program's image. This name often comes with the actual version of the library in it, so that if the executable is moved to a different environment where a compatible library's version is not present, the loader fails to resolve any call, so as to avoid undefined behaviours;

2. it reserves an entry in the *Procedure Linkage Table* (PLT), for each library function the code references. Any call to this function will refer the associated PLT entry, which keeps enough space to host a couple of machine instructions;

3. for each entry in the PLT, it reserves the corresponding entry in the *Global Offset Table* (GOT), which stores the memory address of the function's entry point.

The need for these tables arises from to the *lazy binding* policy adopted by the dynamic loader. PLT and GOT tables reference each other so that the system can know whether the symbol associated to a specific library function is being called for the first time or where it is located, otherwise. In the former case, the library symbol is resolved, otherwise the (already-resolved) function's symbol is simply referenced by a call. A very high-level graphical overview of this is given by Fig-

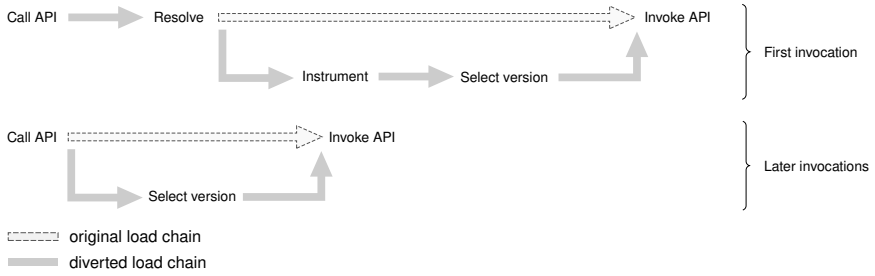
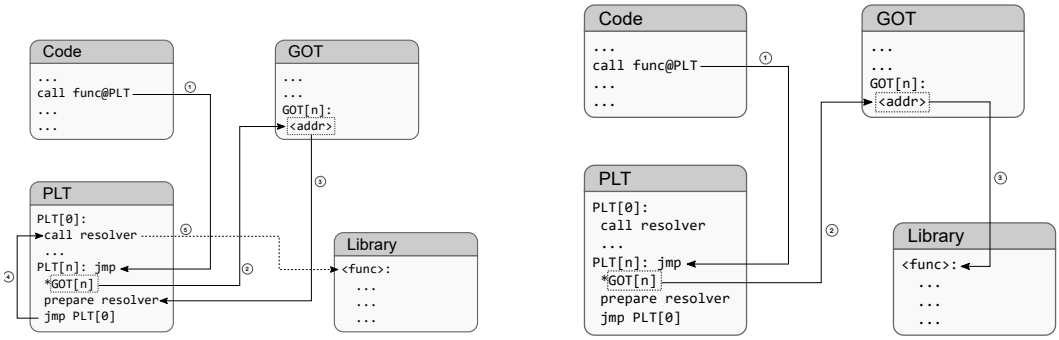


Figure 5.2. High-level dynamic symbol's address resolution flow for the shared objects.

ure 5.2, that briefly depicts the standard runtime resolution of a shared symbol in conventional systems. It is exactly where our proposal acts so as to generate a reversible-enabled version of third-party library functions.

Let us proceed illustrating the symbol resolution mechanism, by taking advantage of some graphics, and suppose that a program relies on the shared library's function `func`; hence, GOT and PLT tables are organized as in Figure 5.3a. The generic call to `func` actually translates into a call to the n-th entry of the PLT table `PLT[n]`, where `n` is the index associated with shared library function `func`. The reader may have already observed that the first PLT table entry, namely `PLT[0]`, is a special entry reserved to a call instruction to the `resolver` function, which represents the dynamic linker in charge of determining where the entry point of the library function is actually loaded in memory. By PLT construction, each entry of this table has an indirect jump to the address held by the corresponding `GOT[n]` entry, which is initialized with the address `PLT[n] + 8`; namely, the instruction just after the indirect jump of the relative `PLT[n]`. Once the very first call to `func` is issued, the execution flow is redirected toward the code in `PLT[n]`, which takes control. At this point, the execution flow is at PLT[n]+8 and the subsequent code snippet prepares (on the stack) a set of parameters before jumping to `PLT[0]` and give control to the runtime environment's symbol resolver `resolver`. In its turn, the resolver will evaluate the effective address associated to the library function demanded (i.e `func`) and will place it permanently into the corresponding `GOT[n]` entry for subsequent references. Nonetheless, for the first invocation, this function traverses a complex call stack of accessory system functions; basically it (*a*) maps the shared object in the virtual address space, making it accessible to the caller application, and (*b*) seeks the desired library function, within it. As the resolver terminates successfully (i.e., the library function has been found), it updates the `GOT[n]` entry with `func`'s address and returns the control with a jump to that address. The GOT/PLT organization, after the symbol's resolution, is depicted in Figure 5.3a. Hereafter, any other future call to `func` will not cause the activation of the dynamic resolver, as the address stored in `GOT[n]` now references the effective address of `func`.



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

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

Figure 5.3. Dynamic symbols resolution procedure relying on the GOT and PLT tables.

The above described procedure is the system default path followed by the system resolver, however we need to divert this normal flow toward a custom block of code. This code is in charge of creating a reversible version of the library function to provide to the upper application, transparently. Again, even in this case we employed the instrumentation technique, by extracting a compact version of the Hijacker's kernel so as to make the applications "self-instrumentable". A digression of how we do this is provided in the following Section 5.1.1 "*Intercepting dynamic linker's resolver*".

5.1.1 Intercepting dynamic linker's resolver

In order to instrument third-party library function calls, we specifically intercept the above-described mechanism, by means of our reversing framework (Chapter 3) that relies on the already mentioned library manager. This module contains a *program constructor*, namely a function which is activated by the program loader before giving control to the actual `main` program. The goal of this constructor is to replace the call to `resolver` to a different function, exposed by the library itself, which alters the behaviour of the latter part of the dynamic linking process. In particular, the custom resolver takes the following steps:

1. similarly to the dynamic linker's resolver, it determines what is `func`'s entry point virtual address;

2. once `func`'s address is identified, it creates a copy of the whole function in memory, instrumenting any instruction which has a memory operand as the destination (namely, a *memory-write* instruction);

3. the instrumentation is carried in a way such that before executing the actual memory write operation, control is given to a *trampoline* which activates a module of `libreverse`;

4. an entry in a custom table, called *Library Activation Trampoline* (LAT), is reserved. This entry keeps a small portion of code to determine whether the instrumented version of the library should be called or not;

5. the address of `LAT[n]` is stored into `GOT[n]`, allowing any future activation of `func` to directly give control to the code in `LAT[n]`;

6. control is given to `LAT[n]`, in order to perform the actual function call.

All these points above demand special care, and we therefore describe in the following of this chapter. This general scheme allows to intercept *any* call to *any* function in *any* third-party shared library. Therefore, it is also eligible to support rollback operation since aware of the reversibility requirements of Time Warp-based simulations.

Although the steps taken by the dynamic linker's resolver are mostly standardized, there could be some variability across systems and versions of the linker regarding the actual steps taken. To make our support of general availability, we want our custom resolver to execute the same steps as the system's dynamic linker does. To ensure portability across systems and linker versions, we leverage the conventional program startup path of Figure 5.1) to clone the system resolver and patch it according to our needs. When the program is launched, a hook to our reversing framework's constructor is activated (the light-grey highlighted step of Figure 5.1), it replaces in `PLT[0]` the address of the resolver with a custom one. Nevertheless, since this custom resolver should be compliant with the system's one, `libreverse` does not contain the custom resolver's code itself, rather, it generates the code at program startup by creating a copy of the system's resolver, except for adding some additional logic. To understand how it works, we need to introduce first what standard dynamic linker's resolver has to perform. First, it determines whether the image of the shared library is mapped into the program's image—in the case of an ELF format this is the `.dynsym` section,—if not it has to be `mmap`'ed to be usable by the application. Once the library is ready and mapped, the resolver locates the function's entry point within the library im-

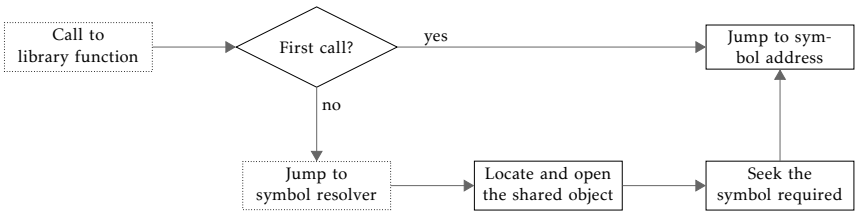


Figure 5.4. Flowchart of the dynamic symbol resolver.

age; this is commonly done via a mechanism based on a fast hash function, which relies on data stored in the program's image. At this point, the resolver can actually resolve the effective address of the required function and store it in relative `GOT[n]` entry, where the index `n` has been passed as an argument on stack (i.e. the `prepare_resolver` phase depicted in Figure 5.3). Finally, once the address is ready, the standard resolver returns control by invoking the library function. The whole procedure flowchart is presented in Figure 5.4.

It is just before returning control to the actual library service where we divert the conventional flow toward our custom hook. To let the reader understand the approach, we have to recall that the activation of a library's function is made via an *indirect jump*. On x86 systems, this is implemented by an instruction in the form `jmp *%reg`; the address of the target function is stored into a register `reg` used as the destination operand. Once our constructor takes control, it creates a copy of the system resolver's code, and starts scanning its bytes until such an instruction is found. At this point, it is then replaced with a *direct jump* to the library manager, which implements a compact instrumentation kernel, in charge of instrument at runtime a copy of the demanded function just before the actual library invocation. The instrumented version of the library will be mapped as part of the program's image, analogously to what the system normally does. Through the library manager, we can deviate the invocation flow, instrument a clone copy of the library's service and choose which version to effectively run, whether the original or the instrumented one, just by means of the LAT table, which holds the address of both. This strategy allows `libreverse` to attach its library manager to any version of the dynamic symbol resolver, independently of the actual way the identification of the function's symbol within the shared library's image is carried out. So far, we described how the dynamic linker is "patched", let's hence move on to explaining what happens inside the library manager's instrumentation function.

5.1.2 Instrumentation of library functions

Intercepting the dynamic linker, we can take control right after the resolution of the function's address; however we must determine its size, before to actual instrumenting the function. Since the executable keeps track of the library file on disk,

the library manager inquires the system to retrieve the information needed, i.e., the symbol's metadata and starts the instrumentation process. We recall that this process entails adding a portion of code for any relevant instruction that writes on memory (Section 3.1.3 "*The reverse code engine*") and inevitably generates an instrumented version which is larger than the original one. For this reason, we allocate a memory area of double the original size via the `mmap()` system-call, making it both writeable and executable. We then copy the whole content of the functions' binary representation in the mapped area; this will be the working copy, which we can inspect and alter privately.

The instrumentation process of shared library symbols entails two *logical steps*. The first one requires determining the total number of assembly instructions which compose the function; the second one is identifying the instructions which write on memory. As widely described in Chapter 3 "*The reversing framework*", these instructions should be properly altered to generate *reverse instructions* on the fly, arranged into a reverse window (Section 3.1.1 "*The reverse window*"). We note that these two steps involve two different levels of detail and, thus, complexity. Indeed, to determine the number of instructions, we do not need to get down into their *semantics*. This is a non-minimal optimization on the x86 architecture. The x86 ISA allows a variable-size one of the instruction binary encoding, meaning that the length in bytes of a single assembly instruction is unknown beforehand. Only by interpreting the *opcode*[2], it is possible to determine the exact number of instruction's parameters, thus its length. For the sake of performance, `libreverse` is equipped with two different disassemblers. The first one—we call it a *length disassembler*[3]—is a fast table-based routine. It only tells what is the length of the actual instruction in bytes and gives a reference to the actual opcode[4]. The second disassembler included into `libreverse` is a *full disassembler*. It decodes the whole instruction's bytecode, populating an accessory structure that describes the instruction's semantic. The execution of the length disassembler is 3 times faster than the full disassembler on any x86 instruction, regardless of its length. As a side note, we refer to the instruction's bytecode as its binary representation encoded in the object file format, namely the ELF file in our case.

The library manager enacts the instrumentation process in the following way. The length disassembler is invoked on the initial address of the function, returning the size of the first instruction and a pointer to its actual opcode, which is matched against a table telling whether the instruction *could* entail a memory-write opera-

[2]Please, refer to [64] for a description of the x86 instruction's format.

[3]The source code of the length disassembler (`lend`) is available at `https://github.com/HPDCS/libreverse/tree/master/src/lend`

[4]In fact, x86 instructions' format allows an arbitrary number of prefixes to precede the instruction's opcode. As a result, the first byte in a given bytecode representation is not necessarily the opcode.

tion. In the negative case, the instrumentation process continues by scanning the next instruction identified by inspecting the bytecode located n bytes after the initial address, where n is the length returned by the length disassembler. On the contrary, the full disassembler is invoked. On the one hand, it first ensures the instruction is actually a memory-write and, on the other hand, it extracts the *size* (in bytes) and the *destination address* parameters necessary to generate the inverse instruction (we redirect to Section 3.1.3 "*The reverse code engine*" for more details). The instrumentation process *replaces* the memory-write instruction with a *jump* to a code snippet generated at runtime. This snippet resembles the *trampoline hook* we discussed in Section 3.1 "*Architectural details*", but it is a slightly modified version, conceived to comply with the different nature of the instrumentation process we adopt here. Notwithstanding, this trampoline code snippet still keeps and prepares the required information to generate the associated inverse instruction, by calling the well-known *reverse code generator* (Section 3.1.3 "*The reverse code engine*"), analogously to what described throughout the Chapter 3 "*The reversing framework*". The library manager places the trampoline snippet into an additional executable memory area, however the number of memory-write instructions is unknown beforehand. The memory area assigned to trampoline snippets is preallocated and keeps the space only for a certain number of trampolines; therefore if space exhausts, a new memory area is silently allocated.

The trampoline's code is organized as in Listing 5.1. The first required action is to save the CPU context because the original library function is (and actually must be) unaware of any injected code's execution. Unfortunately, since the code was placed *after* the program's compilation, standard `setjmp/longjmp` functions cannot be used, because we are explicitly breaking System V ABI's calling conventions [2, 3]. Hence, we need to save all the so-called *caller-save* registers manually. Line 1 of Listing 5.1 does a fast CPU-context save by pushing all required general-purpose registers and the flags register. Since the code is crafted directly in assembly language, we do use only *caller-save* registers. Registers used by functions called by the trampoline is not a concern, as their code is compiler-generated and therefore respects the calling conventions, preserving the consistency of the program's execution. Guaranteed the CPU context is saved, we can call the same reversal engine `reverse_code_generator` seen in Section 3.1 "*Architectural details*", which computes the target memory-write address and generates the corresponding reverse instruction.

The original instruction replaced by a jump to the trampoline is embedded into the related snippet, just after the call to the `reverse_code_generator`. Nevertheless, the original instruction might require contextual information in order to execute correctly, because many instructions in the x86 ISA use *relative* references. As an example, consider an operation used to store a value into a *local*

```
1 save CPU context (except RIP register)
2 call reverse_code_generator
3 restore CPU context (except RIP register)
4 <original instruction>
5 jmp <address>
```

Listing 5.1. The instruction trampoline.

variable. These variables are stored on the stack and are often referenced using a displacement from either the base frame pointer or from the stack pointer. Therefore, before giving control to the copy of the memory-write instruction we need to restore the CPU context (Line 3 of Listing 5.1), except for the value of the RIP register (the program counter). The choice of excluding texttttRIP allows to correctly execute a large set of instructions, although we must explicitly account for the fact that the value kept by RIP is different from the original execution context. This latter point deserves an additional discussion. Indeed, the 64-bit version of the x64 architecture supports the RIP-relative addressing mode. It is a particular addressing mode that allows to target symbols (e.g. variables) by encoding a displacement from the current value of the RIP register. Shared libraries heavily use the RIP-relative mode, as an effective way to implement the so-called *position-independent code* (PIC), to guarantee shared objects to be remapped to any virtual address; in the 64-bit x86 ISA, this entails a massive usage of the RIP-relative addressing mode. To our goal, overcoming RIP-relative issue does not translate to a mere restoration of the whole CPU context, including the value of RIP. In fact, at the original address, we no longer have the original instruction. To execute its copy RIP *must* point to the copy, which is at a different address. Therefore, to correctly execute memory-write operations which rely on RIP-relative addressing, the only option is to *fix* the displacement. To this end, we rely on the length disassembler. In particular, this disassembler sets a global (per-thread) flag whenever it encounters an assembly instruction which is using the RIP-relative addressing mode. Once such an instruction is found, the full disassembler is invoked on it, allowing to determine whether this addressing mode is used in the source or in the destination operand. In both cases, the instrumentation process will realign the displacement to the correct offset. This correction is trivial: we can at any time determine what is the *additional offset* (either positive or negative) introduced by the instruction shifting to a different location due to the instrumentation process. Nevertheless, we create a whole copy of the original library function, so the correction of RIP-relative addressing cannot be limited to instructions copied into the trampoline, rather, all RIP-relative addressing must be corrected. Besides, by rely-

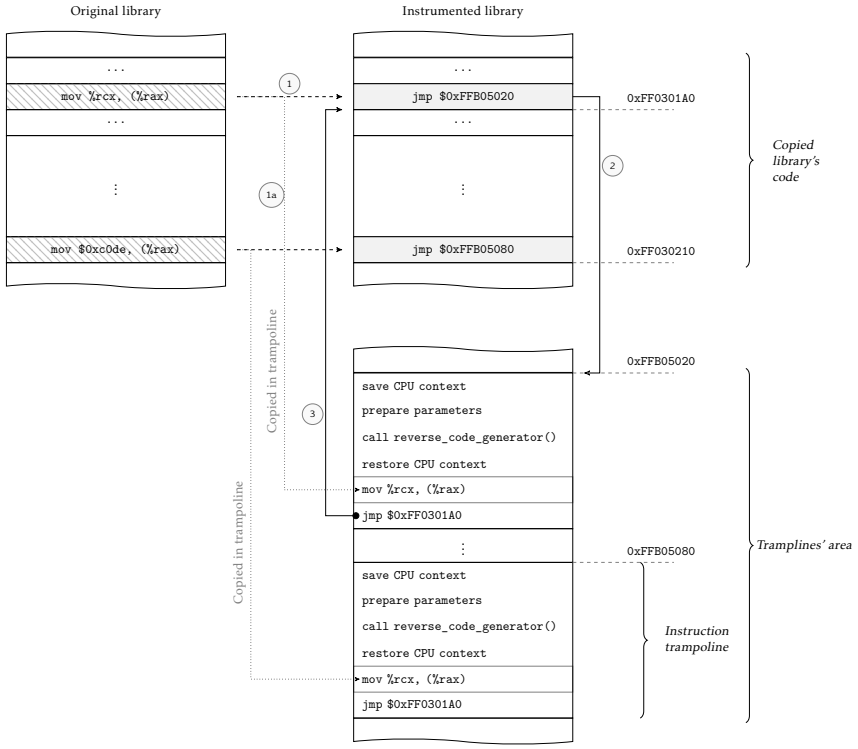


Figure 5.5. First the memory-write instruction is replaced by a direct jump to trampoline (1); this jump points to the runtime-generated trampoline snippet that contains the original instruction (1a); the jump points to te trampoline's snippet (2), and finally once generated the inverse instruction trampoline's code return control to next instruction of the library's function (3).

ing on the above-described scheme, we can apply the adjustment in-place during the copy of the instructions.

To complete the instrumentation process of the library function, we iterate over all the instructions carrying on the aforementioned steps, until we reach the end of the function. As mentioned, we can identify the end of the function by inspecting the library's ELF symbol table, to determine its total length in bytes. The final instruction of the instruction trampoline must give control back to the library function. Since there is a one-to-one mapping between the memory-write instruction and the entry in the relative trampoline snippet, the return from it is realized with a direct `jmp` instruction. The landing address in the library's code is the one just after the `jmp` instruction that replaced the original memory-write. The whole process hitherto described is illustrated in Figure 5.5.

```
1 mov     %fs:_dso_mode@tpoff,%eax
2 cmpb    $0x0,%eax
3 jz      1f
4 call    original_function
5 ret
6 1: call    instrumented_function
7 ret
```

Listing 5.2. Entry of the LAT table (x86 64-bit version).

After that the whole function is instrumented, it remains to hook the altered version to the GOT/PLT invocation mechanism. As hinted before, we want to give the possibility to activate both the original version and the instrumented one, depending on the execution context. In particular, the reversibility facilities are only related to the execution of the simulation models' event handlers, while when executing in *platform mode* (i.e. when the control is taken by the simulation engine) we do not need to generate any reverse instructions. In this latter case, for the sake of performance, we want to rely on the original version of the library functions. To allow a fast switch between the two versions, we rely on the aforementioned LAT table. In particular, the n-th entry in the LAT, which corresponds to the current function being instrumented, is organized as in Listing 5.2 for a total of 24 bytes[5]. The goal of this code is to check the (per-thread) global variable `_dso_mode` which holds the execution mode required for the invocation; namely, whether a library function is called (*a*) from the simulation engine or (*b*) from the application code. In the former case, the framework routes the call to the original (non-instrumented) library function; in the latter case, on the contrary, it invokes its instrumented version. To change the execution mode, `libreverse`'s library manager offers the internal API function `switch_operating_mode(int flags)`—Table 5.1 gives a summary of the possible operating modes supported and their relative use. From a practical point of view, we use this API to tell the system whether the control will pass to an event handler or it is returning from such a handler. It is the responsibility of the simulation engine to properly use this function, according to its needs. Overall, the integration with the GOT/PLT invocation mechanism is simply done by placing the address of crafted n-th LAT's entry within the corresponding n-th one of the GOT table. Figure 5.6 illustrates the whole instrumentation process we described so far.

Libraries are often considerably optimized and nothing prevents to somehow "break" the conventional idea of function, in terms of execution flow—with "conventional idea of function" we refer to the execution confined within the bound-

[5]We, again, refer to the case of 64-bit x86 Linux systems.

Name	Value	Description	Environment
`MODE_PLATFORM`	`0`	Tells the API to switch to the platform execution mode, namely the one used by the simulation engine. The `libreverse` framework will route library calls toward the *original plain* version.	Simulation engine
`MODE_REVERSIBLE`	`1`	Tells the API to switch to the reversible execution mode, namely the one application code relies on. The `libreverse` framework will route library calls toward the *instrumented* version.	Application code

Table 5.1. Definition of the possible `libreverse`'s execution modes.

aries of the related symbol's size. This is a typical scenario happening, for instance, in `glibc` where one function might jump into the middle of another one, just to execute a portion of its code—albeit it might sounds strange, it happens under specific conditions the host system can detect. While this scenario can be easily detected while resolving code references as soon as they fall outside the function's symbol, handling this condition is less trivial, as it would entail some code flow analysis like the ones presented in [42]. Since such an analysis is out of the scope of this thesis, and considering that a library as complex as `glibc` shows this behaviour only in a handful of functions (like, e.g., `memmove()`), for the sake of simplicity we replace these functions with less-optimized ones which are statically linked to the executable. Future work entails generalizing the approach, in order to specifically deal with this corner case with any third-party library.

So far, we described how the instrumentation process patched the GOT and PLT system tables and how library's function calls are delivered to the proper endpoint, according to the required operating mode; however there are some other side aspects we have to handle and discuss. As for the linkage the instrumented library's code with the corresponding trampoline code, a `jmp` instruction replaces the original memory-write one. Nevertheless, it may pose an additional problem due to the variable instructions' length of the x86 ISA. Two different cases may happen; if the size of the `jmp` instruction (5 bytes) is smaller than the size of the actual intercepted memory-write instruction, the remaining space can be easily filled with a number of `nop` instructions. Conversely, the memory-write instruction's might be shorter than 5 bytes. We cope with this case "making room" for the jump to the trampoline by coalescing multiple consecutive instructions in the same trampoline code. In this case, the disassembly of the library function continues until enough space for the `jmp` is found; if the process reaches the end of

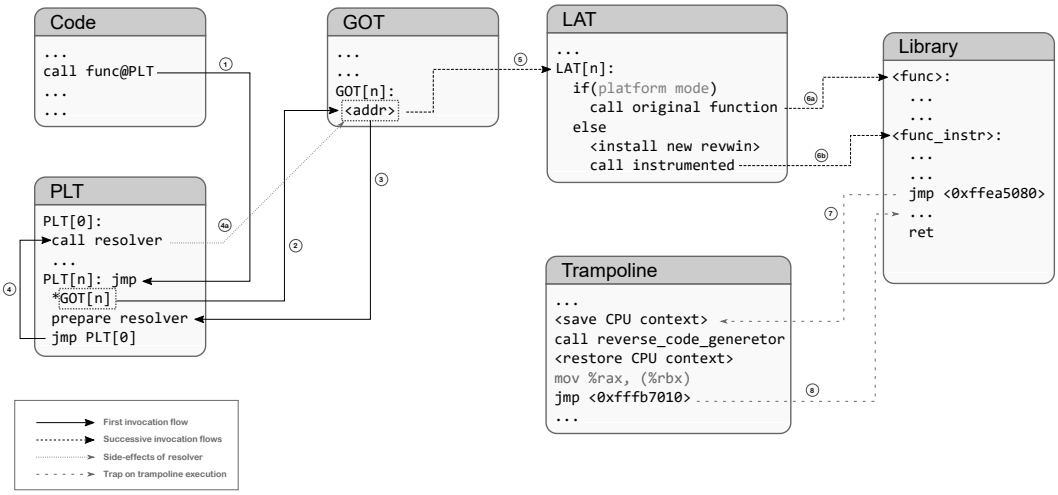


Figure 5.6. Organization of code, tables, and trampolines after instrumentation. The application invokes the library function `func`, by calling into `PLT[n]` (1); initially GOT[n] holds the resolver's address (2), which is invoked (3) and (4); the resolves replace the address in `GOT[n]` (4a); then, the program jumps to the corresponding `LAT` entry by dereferencing the n-th pointer in the `GOT` ((5)); according to *platform mode* value, the trampoline activates the original version of `func` (6a), or the instrumented one (6b), otherwise. In case of step 6a the instrumented instructions is moved to the trampoline are accessed via jumps (7); the instrumented function regains control (8).

the function before finding enough room, the execution is "backtracked" to coalesces instructions before the one we are substituting. Anyhow, since the length of an assembly instruction is variable, it could be resource intensive to perform this latter action. To this end, while performing the forward instrumentation, the instrumentation engine builds an *instruction index*. This index has an entry for each instruction and keeps, among others, its size in bytes. Since the number of instructions that compose the function is not known beforehand, the instruction index is implemented as a wait-free resizeable array, as described in [37]. Let us suppose to have the code scenario as:

```
1 jmp 1f
2 movl $0x0, %eax
3 movsl
4 1: leave
5 ret
```

The instrumentation process will detect that the `movs` (Line 3 of Listing 5.1.2) is a memory-write instruction, and will trigger the replacement with a `jmp`. Since `movs` is only 1-byte long, the coalescing procedure will try to expand over subsequent instructions. The next is the 1-byte long `leave`, so the coalescing procedure continues, until the end of the function is reached. At this point, since the total

amount of bytes found amounts to three, the coalescing procedure inspects the instructions' index to determine how many instructions behind the `movs` should be taken to make enough room to the `jmp`. Since the `movl $0x0, %eax` is 5-byte long, the coalescing procedure takes it and halts. This gives a grand total of 8 bytes, sufficient to place our `jmp`. Nevertheless, this action will break the functioning of the program. In fact, the initial instruction in the example is a `jmp` which targets one of the instructions which will be moved into the trampoline entry, having the `jmp` target the middle of the (newly-inserted) assembly instruction. To overcome this issue, we extend the aforementioned instructions index, adding a reference for each instruction to a different instruction (e.g., the case for a `jmp` instruction). Whenever an instruction is moved to a new trampoline snippet, the referencing instruction's offset is corrected, simply applying the corresponding shift to the displacement.

We recall, as a last note, the digression in Section 3.3 "*Dealing with memory allocations*". As for the handling of the memory management, `libreverse` offers two API functions for wrapping the standard allocation and deallocation operations, namely `register_alloc()` and `register_dealloc()`. These functions accept as argument a function pointer each: `void *(*allocate)(void *ptr)` and `void (*deallocate)(void *ptr)`, respectively. These pointers allow to bridge the internals of `libreverse` with the simulation engine's memory manager, so that whenever a library allocates some memory a call to the `deallocate()` function is placed within the reverse window, while when a chunk of memory is deallocated, a call to `allocate()` is similarly stored. We emphasize that having the `allocate()` function accept a pointer is a strategic choice to allow piece-wise-deterministic replay of events upon a rollback operations, allowing to retrieve buffers at the same virtual addresses, and therefore support a memory map laid out in a generic way.

5.2 Experimental assessment

In this section we present the experimental results deriving by the integration of the presented work into the simulation engine ROOT-Sim. In the following Section 5.2.1 "*Test-bed environment*" we introduce the working context we set in for the this specific work. Thus, in Section 5.2.2 "*Test-bed application*", we present and comment the experimental results. As for the reference PDES platform, we still used ROOT-Sim to assess the experimental data of this proposal (refer Section 4.3.1 "*Test-bed platform*").

Parameter	Value
RAM	64GB
Cores	32
Worker threads	32
Operating system	Linux Debian 6
Linux kernel	2.6.32-5-amd64
Architecture	64 bits

Table 5.2. Configuration of the environment for the test-bed application to assess the performance result of `libreverse`.

5.2.1 Test-bed environment

Our experiments have been run on top of a 32-core HP ProLiant server equipped with 64GB of RAM and running Debian 6 on top of the 2.6.32-5-amd64 Linux kernel. This is a common setup for HPC applications, as this is a software configuration which offers a very good tradeoff between the services exposed to user space applications and performance. Nonetheless, our approach is general and can be used as well on more modern environments. A summary of the environment configuration used to assess the strategy discussed so far is presented in Table 5.2.

From a practical point of view, to integrate `libreverse` with ROOT-Sim, we have to cope with two aspects: (*a*) the models' compilation toolchain (which relies on the `rootsim-cc` custom compiler), and (*b*) the reversible computing facilities already developed and plugged in the engine (refer to Chapter 3 "*The reversing framework*"). As for the compilation toolchain, we statically link the final executable against `libreverse`. As the name suggests `libreverse` is developed and compiled as library object itself. The result is that the program constructor described in previous Section 5.1.1 "*Intercepting dynamic linker's resolver*" takes control before the actual simulation engine starts. We recall that the constructor is in charge of patching the dynamic linker. Nevertheless, we already integrated the reversible computing support described in Chapter 3 "*The reversing framework*" into ROOT-Sim, as in the work presented in Chapter 4 "*The hybrid rollback strategy*". Leading from the static binary instrumentation of the reversing framework described, we further enhance the simulation system capabilities by simply plugging `libreverse` into it. While the approach described throughout this chapter is in charge of intercepting calls to external library and properly instrument them, the "static" reversing framework handles any reversible-enabled application event's effect on memory.

To ensure that the reverse execution is performed in the proper order, we rely on the `get_mark()` API exposed by `libreverse`, so that reverse code generated by ROOT-Sim can be tagged with data which allow to determine the total order of

reverse actions to take. In this way, upon a rollback execution, ROOT-Sim is able to undo the effects on memory by the forward execution of events by either relying on its internal reversibility management, or invoking the `execute_revwin()` function.

5.2.2 Test-bed application

As a test-bed application, we rely on the Sensors Network Model (SNM), which simulates the behaviour of wireless sensor networks (WSN). WSN are networks of small devices that comprise a set of simple components like a power source, a microprocessor, a wireless interface, a generic amount of memory and one or more sensors. They are used to gather information in a given location or region, according to the sensor they are equipped by. Due to the limited radio communication range, nodes can communicate relying on multi-hop routing protocols.

SNM implements the so-called Collection Tree Protocol (CTP) [58] to collect data from wireless sensors networks. In particular, it relies on a variant of the library offered by TinyOS [32]. CTP is a distance vector routing protocol, which computes the routes from each node in the network to the root (specified destinations) in the network. Each node forwards packets to its parent, chosen among its neighbour nodes. In order to make a choice, each node must be aware of the state of its neighbours: that's why nodes continuously broadcast special packets, called *beacons*, describing their condition. The metric adopted in CTP for the selection of the parent node is the *Expected Transmissions* (ETX). A node whose ETX is equal to n can deliver a data packet to the root node with an average of n transmissions. The ETX of any node is recursively defined as the ETX of its parent plus the ETX of its link to the parent; the root node represents the base case in this recursion, and its ETX is obviously equal to zero.

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

It is interesting to note that the routing engine has to maintain a table, called routing table, where it stores the last ETX value read in the beacons from each neighbour. In this way, it is able to always choose the "best" neighbour (the one with the lowest multi-hop ETX) as parent. Therefore, it has to continuously update the table reading the information contained in the beacons received from the neighbours. In the simulation model, each LP represents a wireless sensor. The

routing table managed by the routing engine (along with other data structures related to all three components of the CTP protocol) is kept within the LP's simulation state. Upon the reception of a simulation event representing a beacon, the simulation model passes the received information to the CTP library, which recomputes all parameters related to the network and then updates the routing table in the LP state, thus performing a set of memory updates. These memory updates are intercepted by `libreverse`, which therefore enables for reversibility the CTP library.

5.2.3 Performance data

In order to assess the overhead introduced by our proposal, we have relied on experiments using two variants of the model. In one variant, the CTP algorithm is dynamically linked to the simulation model, while in the other, the library's code is directly incorporated into the simulation model. In this latter configuration, the actions to correctly restore a previous checkpoint are completely demanded from ROOT-Sim. As mentioned, ROOT-Sim supports the rollback operation via both checkpointing facilities and reversibility facilities. Therefore, in the experiments we present data related to reversibility obtained when having the CTP library linked against the executable (LIB in the plots), by relying on the reversibility facilities offered by ROOT-Sim (REV in the plots), and by relying on traditional sparse state saving (CKPT in the plots), with the checkpointing interval optimized according to the results in [86].

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

Figure 5.7, we report experimental data when the model is run using 300 sensors. By the results, we can see that the configuration presenting the best performance profile is the one associated with reversible execution managed natively by ROOT-Sim. This is mainly due to the fact that the memory-update profile associated with this configuration does not entail a large number of memory updates.

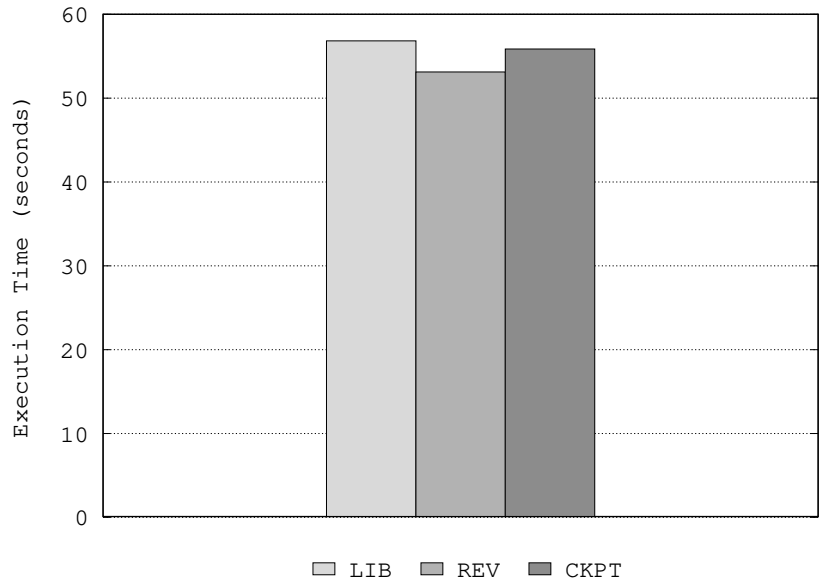


Figure 5.7. Performance data with a workload configuration of 300 sensors.

When using `libreverse`, there is an overhead around 7%, while when relying on checkpointing there is an overhead around 4%, which are both anyhow negligible.

The overhead introduced by `libreverse` is mostly related to the fact that the simulation model performs memory updates as well. Therefore, since the amount of data written by the CTP shared library is not very large, the time spent by the simulation engine to switch among the two reversibility approaches in order to guarantee the correct order of the actions is not paid off. At the same time, the overhead of `libreverse` is slightly higher than that of the checkpoint-based state restore exactly because both the size of the state and the amount of data updated in it is reduced, thus optimized checkpointing facilities (especially is realized according to autonomic facilities as in [86]) are able to fine tune themselves significantly. In addition to this, the rollback length is reduced—sensors are sparse, the communication is organized according to a tree, and therefore the probability of cascading rollbacks are small, and the number of LPs that can rollback each other is limited. In this scenario, plain reverse computation is not surprisingly delivering a better performance.

Figure 5.8 reports experimental data when running the configuration with 2000 sensors. In this configuration, the results slightly change: reverse computation managed by ROOT-Sim still has the better performance result, checkpoint-based executions have the highest performance penalty (more than 12%), while `libreverse`-based reversibility has a overhead slightly smaller than 10%. The trend inversion among checkpoint based and reversibility based is due to the fact that, since the amount of state updates is non minimal, the checkpointing system

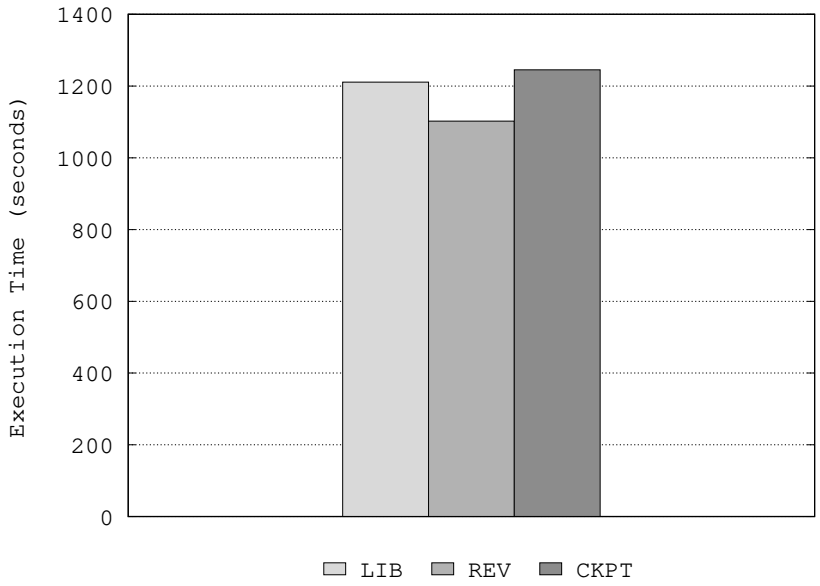


Figure 5.8. Performance data with a workload configuration of 2000 sensors.

has to restore a large amount of data, either in a full or incremental fashion. At the same time, the performance gain by the reversibility engine internal to ROOT-Sim is again related to the fact that in this case there is no need for the continuous switch to guarantee the correct order of the reversibility actions.

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

Chapter 6

Conclusions and future work

"Yes," said Deep Thought. "Life, the Universe, and Everything. There is an answer. But, I'll have to think about it."
— Douglas Adams, The Hitchhiker's Guide to the Galaxy

In this thesis, we have presented innovative contributions in the area of reversible computing. We remark that our approach to reversible computing detaches from the traditional concept found in the literature. As mentioned, while the traditional approach focuses on the restoration of a generic previous computational state reverting all the computational steps that separate it logically from the current state, we shift the focus on a completely different layer of abstraction moving down to the memory substratum.

As we described, code instrumentation is the enabling factor for an effective, transparent, and compatible implementation of our solution. Nevertheless, we had to address first the development of a reasonably capable instrumentation tool. The subject of this part of the work is Hijacker, which was on a rather early stage of development and has seen a notable improvement of its features, fuelled by the reverse window-based idea of reversible computing. A substantial redesign of some internal components of Hijacker, now, allows to use it for our real goal: instrumenting a generic program to intercept every instruction that accesses any memory location in write mode. The result of this interception is not to produce a log of dirty data, but rather to craft machine code dynamically, which reverts side effects on memory perpetrated by the original instruction. Foreseeing a possible general-purpose perspective in the future development of this approach, we refactored Hijacker to be as much flexible as possible to tailor on the user needs.

This choice gave us the possibility to conceive a combined rollback strategy we settled into the parallel simulation context. Nonetheless, the proposal we described in Chapter 3 can be easily extended and improved for a general-purpose application. Within the context of parallel simulation, we proposed a solution that combines traditional state saving with our flavour of reversible computing based

on the reverse window. In addition to the envisage and realization of the building blocks for the concrete result of integrating it within the simulation system, in Chapter 4 we also propose a mathematical model that paves the way for an autonomic decision subsystem of growing power.

An additional contribution presented in Chapter 5 addresses the non-trivial problem of instrumenting shared objects while efficiently guaranteeing that such alterations impact on every program that is currently running in the system. Leveraging the startup chain of a program, we succeeded at redirecting the conventional execution flow toward our custom logic. This logic temporarily diverts the execution so as to create (at runtime) a separate version of the target shared library to alter in our own fence. As expected, since the instrumentation is no longer *static* before the linking stage of the compilation process (recall Figure 3.14), the instrumentation cost appears higher with respect to what we did in the first contribution. As for the efficiency aspect, we optimized the instrumentation process to our best, leading the final solution to be not penalizing the performance side, while offering radically innovative reversibility support.

Finally, in Appendix A, we just explore another kind of combination of rollback strategy, still applied to the simulation context; the one between our reversible computation and the Hardware Transactional Memory. We are interested in assessing how software and hardware facilities can take advantages from each other. In particular, the work leads from the observation that time advancement of the commit horizon—the barrier that delimitates the *safety* boundary—impacts notably the abort probability of speculative events (i.e. unsafe) that are far from it. As presented in the experimental section, the results are not impressive, even though it has to be considered that the size of the test-bed environment impacts negatively. The reason for that is the preliminary stage of this research branch, born more as an exploration than an intensive assessment of specific objectives. Nonetheless, it demonstrates a rather interesting resilience of the abort probability. We can envisage a potential for a more in-depth assessment in the near future.

Coming back to the core reversing framework presented, we want to remark its potential to become a tool to use "out of the box". Looking back at the path we followed, albeit the starting direction was a specific application context (i.e. parallel simulation), we orthogonally slide towards more general applicability and laid the foundation for a comprehensive framework. Recalling our roadmap in Figure 1.1, we begun from the idea of a compact effective way to trace the application's memory footprint and moved towards this direction passing through the integration in the ROOT-Sim simulator and tackling shared objects reversing. We stress the importance of this last contribution, in our evolutionary path, as it lies partially outside the simulation context, embracing something not directly related to it. Thus, following this trend of exploration we want to move again farther; first we plan

to finalize the `libreverse` component, whose weight on our reversing framework grew significantly, becoming part of its kernel; then we plan to envisage solutions in the context of reversible debugging leveraging the potentialities of the reverse window to address even *post-mortem* debugging, and security by implementing a solution for a "merciful" sandbox environment. Such a sandbox could potentially avoid killing the process, upon misbehaviour, thanks to the ability to rewind its state within its boundaries. We can glimpse a potential application of that in online malware analysis.

Bibliography

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

[2] System V Application Binary Interface, Intel386 Architecture Processor Supplement, 1997.

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

[4] U. A. Acar, A. Ahmed, and M. Blume. Imperative Self-Adjusting Computation. In *ACM SIGPLAN Notices*, page 170. ACM, 2008.

[5] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '02*, pages 247–259, 2002.

[6] U. a. Acar, G. E. Blelloch, and R. Harper. Selective Memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '03*, number January, page 31, 2003.

[7] T. Akgul, V. Mooney, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings. 26th International Conference on Software Engineering*, pages 522–531, 2004.

[8] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. In *Paste*, pages 18–25, 2002.

[9] T. Akgul and V. J. Mooney III. Assembly Instruction Level Reverse Execution for Debugging. *ACM Trans. Softw. Eng. Methodol.*, 13(2):149–198, 2004.

[10] I. F. Akyildiz. Performance analysis of a multiprocessor system model with process communication. *Computer Journal*, 35(1):52–61, 1992.

[11] M. J. Ammer, M. P. Frank, T. Knight, N. Love, and C. Vieri. A scalable reversible computer in silicon ? (November 1997), 2000.

[12] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

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

[14] D. W. Bauer, C. D. Carothers, and A. Holder. Scalable time warp on blue gene supercomputers. *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*, pages 35–44, 2009.

[15] H. Bauer and C. Sporrer. Reducing Rollback Overhead In {T}ime-{W}arp Based Distributed Simulation With Optimized Incremental State Saving. In *Simulation Symposium, 1993. Proceedings., 26th Annual*. IEEE Computer Society, mar 1993.

[16] S. Bellenot. State Skipping Performance with the Time Warp Operating System. In *6th Workshop on Parallel and Distributed Simulation*, pages 53–61, 1992.

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

[18] C. H. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[19] C. H. Bennett. Demons, Engines and the Second Law. *Scientific American*, 257(5):108–116, 1987.

[20] C. H. Bennett. Time/Space Trade-Offs for Reversible Computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.

[21] C. H. Bennett. Notes on Landauer's principle, reversible computation, and Maxwell's Demon. *Studies in History and Philosophy of Science Part B - Studies in History and Philosophy of Modern Physics*, 34(3):501–510, 2003.

[22] A. Bérut, A. Petrosyan, and S. Ciliberto. Information and thermodynamics: Experimental verification of Landauer's Erasure principle. *Journal of Statistical Mechanics: Theory and Experiment*, 2015(6):187, mar 2015.

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

[24] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. *Journal of Physics A: Mathematical and General*, 34(35):6821–6830, 2001.

[25] C. D. Carothers, D. W. Bauer, and S. Pearce. ROSS: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[26] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[27] D. Cingolani. *Application transparent and efficient mixed state-saving in speculative simulation platforms*. PhD thesis, Sapienza, Università di Roma, 2014.

[28] D. Cingolani, M. Ianni, A. Pellegrini, and F. Quaglia. Mixing hardware and software reversibility for speculative parallel discrete event simulation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9720, pages 137–152, 2016.

[29] D. Cingolani, A. Pellegrini, and F. Quaglia. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. In *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 211–222, New York, New York, USA, 2015. ACM Press.

[30] D. Cingolani, A. Pellegrini, and F. Quaglia. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Transactions on Modeling and Computer Simulation*, 27(2):26, 2017.

[31] D. Cingolani, A. Pellegrini, M. Schordan, F. Quaglia, and D. R. Jefferson. Dealing with Reversibility of Shared Libraries in PDES. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 41–52, 2017.

[32] U. Colesanti and S. Santini. The Collection Tree Protocol for the Castalia Wireless Sensor Networks Simulator. *Month*, pages 1–26, 2011.

[33] M. Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15(1):1–40, 2004.

[34] V. Cortellessa and F. Quaglia. A checkpointing-recovery scheme for Time Warp parallel simulation. *Parallel Computing*, 27(9):1227–1252, 2001.

[35] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000.

[36] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[37] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In M. M. A. A. Shvartsman, editor, *Principles of Distributed Systems*, pages 142–156, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[38] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive Imperative Programming with Dataflow Constraints. *ACM Transactions on Programming Languages and Systems*, 37(1):1–53, 2014.

[39] P. Di Sanzo, F. Quaglia, B. Ciciani, A. Pellegrini, D. Didona, P. Romano, R. Palmieri, and S. Peluso. A flexible framework for accurate simulation of cloud in-memory data stores. *Simulation Modelling Practice and Theory*, 58:219–238, 2015.

[40] D. Dice. TLRW: Return of the read-write lock. *Proceedings of the 22nd ACM symposium on . . .* , 2010.

[41] S. Dinesh Kumar, H. Thapliyal, A. Mohammad, and K. S. Perumalla. Design exploration of a Symmetric Pass Gate Adiabatic Logic for energy-efficient and secure hardware. *Integration, the VLSI Journal*, 58(September 2016):369–377, 2017.

[42] S. Economo, D. Cingolani, A. Pellegrini, and F. Quaglia. Configurable and efficient memory access tracing via selective expression-based x86 binary instrumentation. In *Proceedings - 2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016*, MASCOTS, pages 261–270. IEEE Computer Society, 2016.

[43] A. Ferscha. Probabilistic Adaptive Direct Optimism Control in Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 120–129. IEEE Computer Society, 1995.

[44] A. Ferscha and J. Luthi. Estimating Rollback Overhead for Optimism Control in Time Warp. In *Proceedings of the 28th Annual Simulation Symposium*, pages 2–12. IEEE Computer Society, apr 1995.

[45] J. Fleischmann and P. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. *Proceedings of the ninth workshop on Parallel and distributed simulation*, 25(1):50–58, jul 1995.

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

[47] C. Forbes, M. Evans, N. Hastings, and B. Peacock. Erlang distribution. *Statistical distributions*, pages 84–85, 2010.

[48] M. P. Frank. *Reversibility for efficient computing*. PhD thesis, Citeseer, 1999.

[49] M. P. Frank. The physical limits of computing. *Computing in Science and Engineering*, 4(3):16–26, 2002.

[50] M. P. Frank. *Reversible Computing - Toffoli.pdf*. Number January. Springer, 2004.

[51] M. P. Frank. Throwing computing into reverse. *IEEE Spectrum*, 54(9):32–37, 2017.

[52] E. Fredkin and T. Toffoli. Design principles for achieving high-performance submicron digital technologies. In *Collision-based computing*. Springer, 1978.

[53] E. Fredkin and T. Toffoli. Conservative Logic. *International Journal of theoretical physics*, 21(3-4):219–253, 1982.

[54] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, volume 3, pages 1381–1384. IEEE, 1998.

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

[56] R. M. Fujimoto. Performance of Time Warp Under Synthetic Workloads, 1990.

[57] R. M. Fujimoto, K. S. Panesar, and K. S. Panesar. Buffer management in shared-memory time warp systems. In *PADS*, pages 149–156, 1995.

[58] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems - SenSys '09*, page 1, New York, New York, USA, 2009. ACM Press.

[59] B. Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[60] J. S. Hall. An Electroid Switching Model for Reversible Computer Architectures. *Workshop on Physics and Computation, Laboratory for Computer Science Research Rutgers University*, pages 1–9, 1992.

[61] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):289–300, 1993.

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

[63] M. Ianni, R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia. The Ultimate Share-Everything PDES System. In F. Quaglia, A. Pellegrini, and G. K. Theodoropoulos, editors, *Proceedings of the 2018 {ACM} {SIGSIM} Conference on Principles of Advanced Discrete Simulation, Rome, Italy, May 23-25, 2018*, pages 73–84. ACM, 2018.

[64] Intel. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture, 2000.

[65] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, jul 1985.

[66] D. R. Jefferson. Virtual Time II: Storage Management in Conservative and Optimistic Systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 75–89, New York, NY, USA, 1990. ACM.

[67] V. Jha and R. L. Bagrodia. Simultaneous events and lookahead in simulation protocols. *ACM Transactions on Modeling and Computer Simulation*, 10(3):241–267, 2000.

[68] S. Kandukuri and S. Boyd. Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[69] L. B. Kish, C. G. Granqvist, S. P. Khatri, and J. M. Smulko. Critical Remarks on Landauer ' s principle of erasure – dissipation Including notes on Maxwell demons and Szilard engines. (Icnf):1–4, 2015.

[70] R. Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[71] J. M. LaPre, C. D. Carothers, and D. R. Jefferson. Warp speed: executing Time Warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13*, page 327, New York, New York, USA, 2013. ACM Press.

[72] J. M. LaPre, E. J. Gonsiorowski, and C. D. Carothers. LORAIN: A step closer to the PDES "Holy grail". *SIGSIM-PADS 2014 - Proceedings of the 2014 ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, pages 3–14, 2014.

[73] J. M. LaPre, E. J. Gonsiorowski, C. D. Carothers, J. Jenkins, P. Carns, and R. Ross. Time warp state restoration via delta encoding. 2015.

[74] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[75] J. I. Leivent and R. J. Watro. Mathematical Foundations of Time Warp Systems. *ACM Transactions on Programming Languages and Systems*, 15(5):771–794, 1993.

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

[77] R. C. Merkle. *Reversible Electronic Logic Using Switches*. 1994.

[78] K. Morita. *Theory of Reversible Computing*. 2017.

[79] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003.

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

[81] A. Pellegrini. 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, pages 650–655. IEEE Computer Society, 2013.

[82] A. Pellegrini. The ROme OpTimistic Simulator : A Tutorial Discrete Event Simulation (DES). In *Euro-Par Workshops*, volume 8374 of *PADABS*, pages 501–512. LNCS, Springer-Verlag, 2013.

[83] A. Pellegrini, R. Vitali, and F. Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*, PADS, pages 45–53. IEEE Computer Society, jun 2009.

[84] A. Pellegrini, R. Vitali, and F. Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*, pages 45–53. IEEE, 2009.

[85] A. Pellegrini, R. Vitali, and F. Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th ICST Conference of Simulation Tools and Techniques (SIMUTools)*, SIMUTools, pages 96–98. ACM, mar 2011.

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

[87] K. S. Perumalla. Scaling time warp-based discrete event execution to 10^4 processors on a Blue Gene supercomputer. *Proceedings of the 4th international conference on Computing frontiers - CF '07*, (May):69, 2007.

[88] K. S. Perumalla. *Introduction to Reversible Computing*. CRC Press, 2013.

[89] K. S. Perumalla and V. A. Protopopescu. Reversible Simulations of Elastic Collisions. 23(2):1–38, 2013.

[90] K. S. Perumalla and S. B. Yoginath. Towards Reversible Basic Linear Algebra Subprograms: A Performance Study. *Transactions on Computational Science XXIV*, pages 56–73, 2014.

[91] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, 1994.

[92] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148, 2006.

[93] F. Quaglia. Event History Based Sparse State Saving in Time Warp. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pages 72–79. IEEE Computer Society, 1998.

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

[95] F. Quaglia and R. Baldoni. Exploiting Intra-Object Dependencies in Parallel Simulation. *Inf. Process. Lett.*, 70(3):119–125, 1999.

[96] D. Quinlan and C. Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, 2011.

[97] D. Quinlan, C. Liao, J. Too, R. P. Matzke, M. Schordan, and P. H. Lin. ROSE compiler infrastructure, 2012.

[98] R. Rönngren and R. Ayani. Adaptive checkpointing in Time Warp. In *ACM SIGSIM Simulation Digest*, volume 24, pages 110–117, New York, New York, USA, 1994. ACM Press.

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

[100] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat. Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, volume 26 of *PADS*, pages 70–77. IEEE Computer Society, 1996.

[101] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia. HTM Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models. In *Proceedings of the 22nd International Conference on High Performance Computing (HiPC)*, HiPC, 2015.

[102] M. Schordan, D. R. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan. Reverse code generation for parallel discrete event simulation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9138, pages 95–110. Springer, Cham, 2015.

[103] M. Schordan, T. Oppelstrup, D. R. Jefferson, P. Barnes, and D. Quinlan. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application. In *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation - SIGSIM-PADS '16*, pages 111–122, New York, New York, USA, 2016. ACM Press.

[104] S. K. Seal and K. S. Perumalla. Reversible Parallel Discrete Event Formulation of a TLM-Based Radio Signal Propagation Model. *ACM Trans. Model. Comput. Simul.*, 22(1):4:1—-4:23, 2011.

[105] C. L. Seitz, A. H. Frey, S. Mattisson, S. D. Rabin, D. A. Speck, and J. L. A. de Snepscheut. Hot clock nMOS, 1985.

[106] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(April 1928):379–423,623–656, 1948.

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

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

[109] P. Teichmann. Adiabatic logic: Future trend and system level perspective. *Springer Series in Advanced Microelectronics*, 34(1), 2012.

[110] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In *2008 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172, 2008.

[111] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[112] C. Vieri, M. J. Ammer, M. P. Frank, N. Margolus, and T. Knight. A Fully Reversible Asymptotically Zero Energy Microprocessor. *Power Driven Microarchitecture Workshop*, pages 1–8, 1998.

[113] C. J. Vieri. Reversible Computer Engineering and Architecture. (1993), 1999.

[114] W. Vogels. Eventually Consistent. *Queue*, 6(6):14, 2008.

[115] J. Von Neumann. The general and logical theory of automata. *Cerebral mechanisms in behavior*, 1:1–2, 1951.

[116] J. Von Neumann and A. W. Burks. Theory of Self Reproducing Automata, 1966.

[117] D. West and K. Panesar. Automatic incremental state saving. *ACM SIGSIM Simulation Digest*, 26(1):78–85, 1996.

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

[119] S. Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, IL, 2002.

[120] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Proceedings of the 2008 conference on Computing frontiers - CF '08*, CF '08, page 43, New York, NY, USA, 2008. ACM.

[121] T. Yokoyama and R. Glück. A Reversible Programming Language and Its Invertible Self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 144–153, New York, NY, USA, 2007. ACM.

[122] S. G. Younis. Asymptotically Zero Energy Computing, 1994.

[123] M. V. Zelkowitz. Reversible Execution. *Communications of the ACM*, 1973.

[124] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W. F. Wong. How to do a million watchpoints: Efficient Debugging using dynamic instrumentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4959 LNCS:147–162, 2008.

Appendix A

Synergistic hardware and software reversibility

In this appendix we present a preliminary exploration of the possibilities our reversible computation can offer. If in Chapter 4 we explored it in combination with a *non-reversible* rollback support such as state saving; here, we investigate the orthogonal perspective of a synergistic exploitation of hardware-based and software-based approaches to reversible computing. Thus, similar in spirit, to the work described in Chapter 4 "*The hybrid rollback strategy*", in this proposal we enable the speculative PDES engine, representing our case study, to dynamically select the best-suited rollback strategy. On the one hand, relying on HTM facilities, inspired to [101], and on the other hand, relying on the extensively discussed reverse windows-based approach (refer to Chapter 3 "*The reversing framework*"). Since this work is at an early stage, a small environment is used for the assessment, nonetheless it may deserve a more substantial assessment in the future within a scaled up environment.

We start from the consideration that the worth of events speculatively executed in a HTM transactional context depends on several factors. A first issue regards the implementation of this hardware support. The commitment of the transaction needs to check/update causality constraints recorded by a set of metadata; the higher the degree of concurrency in accessing these "shadow" metadata, the higher the likelihood of yielding to conflicts, leading the HTM-based transactions to abort. Further, related to the specific context of PDES, this metadata are updated according to the *commit horizon*'s time progress, indirectly determined by the simulation event with the lowest timestamp. Hence, speculatively-processed events with HTM support that are farther ahead of the commit horizon will meet higher abort probability, as the indirect updates on the shadow metadata is denser, accordingly. This abort probability is even sharper if these updates are issued by recent events, rather then elder ones (i.e. associated with lower timestamps). Finally, the HTM support is limited to transactions whose read/write set fits the

transactional hardware cache available; otherwise a capacity conflict perpetrated by other cores would occur. Hence, for models characterized by events (or phases of events) having large data sets, the likelihood of successfully committing relative HTM-based transactions may be (significantly) reduced.

In our speculative PDES engine, we overcome these drawbacks of HTM supports by dynamically enabling any worker thread to process an event both as an HTM-based transaction, and via a (software) reversible version of the original event handler, to reduce the likelihood of running non-valuable transactions. According to what described Chapter 3 "*The reversing framework*" and further implemented in Chapter 4 "*The hybrid rollback strategy*", this version is transparently instrumented so that to feed at runtime the relative reverse window. Commit success probability of events run with software reversible support is regardless of the commit probability of HTM-based transaction. As a result, the engine can fruitfully exploit computing resources relieving the HTM support of a part of the invaluable work.

At the same time, the coexistence of HTM and software reversible supports may incur in a negative interference. In particular, software-reversible events must not hinder valuable HTM-based transactions. To this end, we introduced a prioritization mechanism allowing HTM-processed events to gain higher priority over the others. In this way, the system guarantees that latter class of events would concurrently access any portion of the data set possibly target by the HTM-based transactions. On the other hand, we still enable inter-LP concurrency by exploiting the so-called *weak causality*, drawn from [95], which allows multiple HTM-based transactions to operate on disjoint data sets within the same LP state. It is noteworthy that in our software reversibility scheme we avoid the usage of checkpoints; rather we embrace the reverse window-based reversible computing approach. As a result, since we reduce the typically large usage of memory require by checkpointing, so does the abort probability due to exceeded transactional cache's capacity.

We developed a new simulation engine with a kernel relying on a single fully-shared queue of pending events, inspired to the share-everything PDES paradigm [63]. The simulation engine has been released as open source software[1], and implements the support for the combined rollback strategy as described further in this section. In Appendix A.3 we also discuss the experimental data to assess the effectiveness of this proposal, running the classical PHOLD PDES benchmark [56] on an Intel Haswell processor, with HTM support, equipped with 4 physical cores.

[1] `https://github.com/HPDCS/htmPDES/tree/reverse`

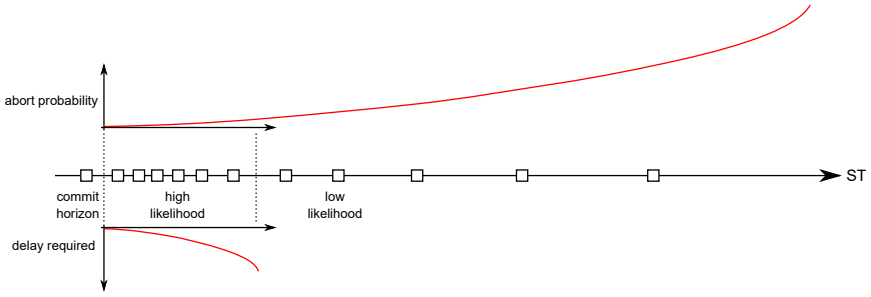


Figure A.1. Three logical regions on the simulation time (ST) axis, with varying density of pending events—those still to be processed, which will possibly generate new ones.

A.1 Simulation horizons and value of speculative work

Introducing our simulation platform, we recall the notion of *commit horizon*. In speculative PDES, we can always identify a point along the simulation time representing the so-called *commit horizon*—commonly referred to as Global Virtual Time (GVT). This horizon is the simulation time instant that distinguishes between events which might be undone, as *unsafe*, and committed events. This time instant can be logically identified by considering that any simulation event e executed at simulation time T can only generate some new event e' associated with timestamp $T' \geq T$. In fact, violating this assumption would imply that an event in the future might affect the past, which is clearly a non-meaningful condition for any real-world process/phenomenon. The commit horizon is trivially identified by the event with the smallest event's timestamp among currently scheduled and processed events, throughout the system—recall that having a single fully shared event queue of pending events avoids at all the problem of in transit events/messages that might affect the determination of the GVT value. In fact, no event still to be executed might produce a causal inconsistency involving the LP in charge of the execution of the commit-horizon event[2].

With our target engine organization, the commit horizon is associated with the oldest event that is currently being executed (or has just been executed) at any worker thread. Therefore, keeping track of the commit horizon boils down to registering, for each worker thread, the timestamp of the event e currently being executed, by replacing the value only after a new event is fetched for processing from the event pool, so that any new event possibly produced by e has its timestamp already reflected into the event pool. The commit horizon can be computed as the minimum among the registered values.

[2]Simultaneous events do not violate this assumption. Nevertheless, if not properly handled by some tie-breaking function [67], they could give rise to livelocks in the speculative execution.

At any time, the commit-horizon event can be considered as a *safe* (namely, causally consistent) one, and therefore does not require any reversibility mechanism for its execution. Let us now discuss about the likelihood of safety of other events to be processed, which stand ahead of the commit horizon. Empirical evidence and statistical considerations based on common distributions for the timestamp increment driving the generation of events in simulation models (see, e.g., [43, 44]) have shown that event patterns are, at any time, characterized by greater density of events, say locality of activities, in the near future of the actual GVT. This situation is depicted in Figure A.1. Also, such locality tends to move along the time axis just based on the advancement of the commit horizon. The implication is that the risk of materialization of causal inconsistencies when speculatively processing one event that is ahead of the commit horizon is somehow linked to its distance from such horizon. This is also linked to the notion of lookahead of DES models, a quantity expressing the minimal timestamp increment we can experience for a given model when processing whichever event that originates new events to be injected in the system. Larger lookahead leads to produce new events in the far future, hence those getting closer to the current commit horizon become automatically safe.

By this consideration, the speculative processing of events that are closer to the commit horizon looks more valuable in terms of avoidance of causality inconsistencies. Hence in our approach we enable the processing of these events as HTM-based transactions, say via the more efficient (lower overhead) recoverability support. We also note that running events that are close to the commit horizon as HTM-based transactions will also lead to a faster advancement of this horizon, as compared to what we would expect if running them via software-based reversibility, since this would lead to longer processing times due to the overhead for producing the undo code blocks. However, an HTM-based transaction can commit only after events standing in the past have already been committed and the corresponding worker threads have already updated their entries in the meta-data array keeping their current timestamp. So, in order to increase the likelihood of committing the HTM-based transactional execution of some event, this transaction typically needs to include a busy-loop delay enabling a waiting phase just before checking whether the meta-data were updated[3]. Checking the meta-data at some wrong point in time will in its turn lead to the impossibility to recheck these data fruitfully in the future, since the updates occurring between the two checks will lead to a data conflict and to the abort of the checking transaction. In Figure A.1 we show how such a delay should be selected somehow proportionally to the distance (in terms of event count) of the event processed via HTM support from the commit

[3]Other kind of delays, such as operating system sleeps, are infeasible since any user/kernel transition will lead an HTM-based transaction to abort deterministically on current HTM-equipped processors.

horizon. Overall, for events that are further ahead from the commit horizon, the delay could not pay off, hence a more profitable approach to speculatively processing them is to run them outside of HTM-based transactions, still with reversibility guarantees achieved via software.

The problem of determining what is the threshold distance from the commit horizon beyond which the HTM support does not pay off is clearly also related to the interference between concurrent HTM-based transactions when using the underlying hardware resources. In fact, if we experience a scenario where two concurrent transactions both require large transactional cache storage for executing the corresponding dispatched events, and the cache is shared across the cores, then even if an event would ideally reveal as causally consistent upon attempting to finalize the transactions, it would anyhow be doomed to abort due to cache capacity conflicts. A similar cache capacity-due abort may even be experienced in case of single HTM-based transaction instance, just depending on the transaction data set, which might exceed the cache capacity. To cope with the runtime adaptive selection of the threshold value, we rely on a hill climbing scheme based on the following parameters, easily measurable at runtime across successive wall-clock-time windows.

T_{HTM} The total processing time spent across all the worker threads while processing events (either committed or aborted) via HTM support.

$COMMIT_{HTM}$ The total number of committed events whose speculative execution has been based on the HTM support.

T_{soft} The total processing time spent across all the worker threads while processing events (either committed or aborted) that are made recoverable via the software-based support (here we include the time spent for instrumentation code used to generate undo code blocks, plus the time for running the undo code blocks in case the events are eventually undone).

$COMMIT_{soft}$ The total number of committed events whose execution has been based on the software support for recoverability.

We compute the so-called work-value ratio (WVR) for both HTM-based and software-based reversible supports as follows:

$$WVR_{HTM} = \frac{T_{HTM}}{COMMIT_{HTM}} \qquad WVR_{soft} = \frac{T_{soft}}{COMMIT_{soft}} \tag{A.1}$$

Equation (A.1) expresses the average amount of CPU time required for performing useful work (namely, for processing an event that is not undone) with

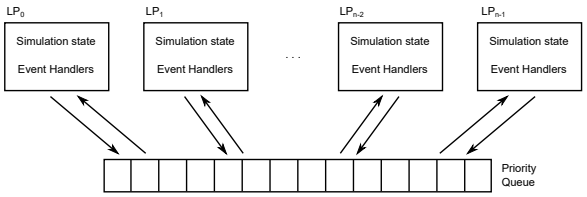


Figure A.2. Basic engine organization

the two different recoverability supports. Then, the threshold value THR determining the commit horizon distance (evaluated as event count) beyond which we consider it more convenient to process the event via software-based reversibility, rather than HTM-based one, is increased or decreased depending on whether the relation $WVR_{HTM} \leq WVR_{soft}$ is verified (as computed on the basis of statistics, on the baseline parameters listed above, collected in the last observation window). In order to avoid stalling in local minima (e.g. due to the avoidance of runtime samples for any of the above listed parameters), we intentionally perturb THR by ± 1 within the hill climbing scheme if its value reaches either zero or the number of threads currently running in the PDES platform.

A.2 The simulation engine's architecture

Before proceeding further to the assessment, we introduce the simulation system designed and implemented for the purpose. We target a baseline speculative PDES engine structure, that is independent of the actual reversibility support adopted, whose architecture is provided in Figure A.2. In compliance with traditional PDES, the engine supports the simulation model partitioning into n distinct LPs, each one associated with a unique identifier (ID) in the range $[0, n-1]$. Each LP is associated with a private simulation state and with one, or more event, handlers realizing the application business logic. The delivery of a simulation event to the correct handler is demanded to the underlying simulation kernel, which is also in charge of guaranteeing the consistency of a shared event pool that keeps scheduled events and of the updates occurring on the LPs' states. As for the event pool, we rely on a shared lock-protected calendar queue [23]. Multiple concurrent worker threads can extract events from the event pool and can concurrently dispatch the execution of the corresponding LPs by activating some event handler as a callback function.

As mentioned, our engine allows the coexistence of hardware-based and software-based reversible computing facilities. Introducing hardware-based reversible computing facilities is somehow easy; recall it can be done using the primitives `TRANSACTION_START`, `TRANSACTION_END`, and `TRANSACTION_ABORT` to drive event processing. Instead, software-based reversibility requires a bit more care,

Algorithm A.1 Shared Lock Acquisition/Release.

```
 1: int lock_vector[n]
 2: double timestamp[n]                                   ▷ To avoid priority inversion
 3: int thread_id[n]                                      ▷ To avoid priority inversion
 4: procedure Lock_LP(e, LP, mode, locking)
 5:     acquired ← false
 6:     do
 7:       if mode = EXCLUSIVE then
 8:         if CAS(-1, 0, lock_vector[LP]) then
 9:           acquired ← true
10:       else
11:         old_lock ← lock_vector[LP]
12:         if old_lock ≥ 0 then
13:           if CAS(old_lock + 1, old_lock, lock_vector[LP]) then
14:             acquired ← true
15:       if ¬acquired then
16:         atomically {
17:           if timestamp[LP] > T(e) ∨ ( timestamp[LP] = T(e)∧ thread_id[LP] > tid) then
18:             timestamp[LP] ← T(e)
19:             thread_id[LP] ← thread_id
20:         }
21:     while ¬acquired ∧ locking
22:     return acquired
23: procedure Unlock_LP(LP)
24:     if lock_vector[LP] = −1 then
25:       lock_vector[LP] ← 0
26:     else
27:       do
28:         old_lock ← lock_vector[LP]
29:       while ¬ CAS(old_lock − 1, old_lock, lock_vector[LP])
```

especially when targeting full transparency to the application-level developer. To cope with this issue, we rely on *static binary instrumentation* as we already described in Section 3.2 "*Addressing code instrumentation*". The instrumented and non-instrumented versions of the application modules coexists by using the mentioned multi-coding scheme, like in Chapter 4 "*The hybrid rollback strategy*". As for the software layer of reversible support, we again rely on our *reverse window* as discussed in Section 3.1 "*Architectural details*".

As mentioned before, to ensure consistency and minimize the effects of data contention on HTM-based execution of events, we must ensure that at no time two different worker threads can execute both software-reversible and hardware-reversible events at once, which target the same LP state. In fact, if this would happen, we might incur the risk of having less valuable work to invalidate more valuable one (since the HTM-based transaction would be aborted if its data set would overlap the write set of the event executed via software-based reversibility). Also, we cannot allow two (or more) events run via software-based reversibility to simultaneously target the same LP state. In fact, these events would not be regulated by any transactional execution scheme[4]. To this end, we rely on a syn-

[4]The undo code blocks guarantee reversibility of memory updates limited to events executing the updates on the LP state in isolation, which complies with classical PDES where each LP is an intrinsically sequential entity.

chronization mechanism similar in spirit to an atomic shared read/write lock [40]. Whenever a worker thread extracts an event from the shared event pool, it first determines whether the event should be executed using hardware-based or software-based reversibility according to the policy introduced in previous Appendix A.1. If the selected execution mode is HTM-based, the worker thread tries to acquire the lock on the target LP in a non-exclusive way, which fails (i.e., requires spinning) in case any other worker thread already took it in an exclusive way. On the other hand, if the selected execution mode is based on software reversibility, the worker thread tries to acquire the lock in an exclusive way, yet this operation requires spinning if at least one worker thread has non-exclusively taken the lock. Nevertheless, this approach might lead to some priority inversion, among the threads which are running more valuable events via the HTM support and threads which are running less valuable events via software-based reversibility. To avoid this, we use a *locking* flag to instruct the algorithm to avoid spinning if it was not possible, for any reason, to acquire the lock—namely, setting *locking* to false transforms the lock into a trylock. If the lock is not taken, two additional values in two arrays are updated atomically: `timestamp` and `thread_id`, which are exploited on a per-LP basis. In particular, the worker thread registers the timestamp it has an event to process at, and its thread id. The latter value is only used to create a total order among threads in case simultaneous events are present, to avoid possible deadlock conditions. These values are periodically inspected by other worker threads (upon a safety check for the current processed event, which fails), so as to determine whether some higher priority event is waiting. In that case, if the work carried out is not likely to be committed shortly, thanks to the reversibility supports it gets squashed, so that higher priority is given immediately to events with a smaller timestamp.

Algorithm A.1 shows the lock management pseudo-code, which relies on the Compare and Swap (CAS) read-modify-write primitive to increase/decrease the value of a shared per-LP counter. The value -1 for the counter means that the lock is exclusively taken, while the value 0 indicates that no thread is running an event bound to the LP. A positive value is a reference counter indicating how many worker threads are concurrently executing events bound to the LP via hardware-based reversibility.

We can now discuss the organization of the main loop of threads within our speculative PDES engine, whose pseudo-code is shown in Algorithm A.2. Essentially, it is made up by three different execution paths, each one associated with one of the different execution modes. Initially, a call to a Fetch() procedure allows to extract from the shared event pool the event with the smallest timestamp. Then, a statistical approximation of the number of events which are expected to fall before the currently fetched event (since others may still be processed or might be

Algorithm A.2 Main Loop.

```
 1: procedure MainLoop
 2:     new_events = ∅                                  ▷ Set of events generated during the execution of an event
 3:     while ¬endSimulation do
 4:         e ← Fetch( )
 5:         if e = NULL then
 6:             goto 3
 7:         events_before ← (T(e) − commit_horizon) / average_timestamp_increment
 8:         if Safe( ) then                             ▷ Safe execution: on the commit horizon
 9:             Lock_LP((e, LP(e), NON_EXCLUSIVE, true))
10:             new_events ← ProcessEvent(e)
11:             Unlock_LP(LP(e))
12:         else if events_before ≤ THR then            ▷ HTM-based execution: high likelihood region
13:             if ¬ Lock_LP((e, LP(e), NON_EXCLUSIVE, false)) then
14:                 goto 7
15:             BeginTransaction( )
16:             new_events ← ProcessEvent(e)
17:             Throttle(events_before)
18:             if Safe( ) then
19:                 CommitTransaction( )
20:                 Unlock_LP(LP(e))
21:             else
22:                 AbortTransaction( )
23:                 Unlock_LP(LP(e))
24:                 goto 7
25:         else                                        ▷ Software-reversible execution: low likelihood region
26:             if ¬ Lock_LP((e, LP(e), EXCLUSIVE, false)) then
27:                 goto 7
28:             SetupUndoCodeBlock( )
29:             new_events ← ProcessEvent_Reversible(e)
30:             while ¬ Safe( ) do
31:                 if timestamp[LP] < T(e) ∨ ( timestamp[LP] = T(e) ∧ thread_id[LP] < tid) then
32:                     Unlock_LP(LP(e))
33:                     UndoEvent(e)
34:                     new_events = ∅
35:                     goto 7
36:         Flush(e, new_events)
37:         atomically {
38:           if thread_id[LP] = tid then
39:             timestamp[LP] ← ∞
40:             thread_id[LP] ← ∞
41:         }
```

produced as a result of the processing) is computed as:

$$\frac{T(e) - commit\_horizon}{average\_timestamp\_increment} \tag{A.2}$$

where $average\_timestamp\_increment$ is computed[5] as $\frac{commit\_horizon}{total\_committed\_events}$. This value, together with the threshold THR (refer previous Appendix A.1), is used to determine whether a certain event might be more valuable or not, thus requiring either HTM-support or software-based reversibility (line 12). Additionally, if

[5]For non-stationary models, where the distribution of the timestamp increment between successive events can change over time in non-negligible way, this same statistic could be simply rejuvenated periodically, by discarding non-recent events commitments and subtracting from $commit\_horizon$ the upper limit of the discarded simulation time portion.

an event is executed exploiting HTM, this value drives as well the selection of a delay before checking again the safety of the corresponding transaction (namely, whether the timestamp of the event has in the meanwhile become the commit horizon), so as to avoid making it doomed with a high likelihood (line 17).

In case of a safe execution, i.e. the execution of the event on the commit horizon (lines 8–11), we take a non-exclusive lock, which is used to inform any other thread that the destination LP is currently processing an event. This avoids that any other worker thread starts processing an event via software-based reversibility at the same LP while we are processing in safe mode. Moreover, we configure the lock to spin because the worker thread in charge of executing this event has the highest priority and any other competing thread will try to give it permission to continue execution as fast as possible.

For a transactional execution (lines 12–24), we use the trylock version of the per-LP lock. If we fail to acquire the lock, the execution resumes from line 7, meaning that we check again whether the extracted event has become safe or not, in the meanwhile. Otherwise, as already explained before, we start executing the event within a HTM-based transaction, introducing an artificial delay—via the Throttle($events\_before$) call—which is proportional to the estimated number of events in between the commit horizon and the currently executed event. If the transaction becomes doomed (lines 21–24) the execution restarts from line 7, so as to check whether the just-aborted event has become safe.

The case of execution via software reversibility (lines 25–35) is a bit different. In fact, first we have to take an exclusive lock—in a trylock fashion, for the same consideration related to the HTM-based execution—and we have to setup the undo code block, by allocating a reverse window buffer. At the end of the execution of the event, similarly to the HTM-based case, we have to wait for the event to become safe. Nevertheless, since this execution entails taking an exclusive lock, we continuously check whether some other thread is registered at the same LP with a higher priority (line 31). This situation might arise due to another event, executed at any other worker thread, generating a new event to the same LP with a timestamp smaller than the one of the event currently processed via software-based reversibility. Failing to make this specific check could either hamper liveness (a thread waits its event to be the commit horizon, which cannot happen) or correctness (events are committed out of order). Line 31, paired with lines 15–20 of Algorithm A.1, is able to ensure both correctness and liveness.

Whenever an event is executed, and then committed thanks to safety assurance, in whichever execution mode, we first place into the calendar queue any possible new event generated (line 36), and we then unregister the thread from the `timestamp` and `thread_id` vectors which are used to avoid priority inversion

(lines 37–41). For the implementations of Fetch(), Flush(), and Safe(), we refer the reader to [101].

A.3 Experimental results

We tested our proposal with the Phold benchmark [55]. We included 1024 LPs in the simulation model, each one scheduling events for itself or for the others. Specifically, upon processing an event, the probability to schedule a new event destined to another LP has been set to 0.2, which is representative of scenarios with non-minimal interactions across the simulated parts. Also, the initial population of events has been set to 1 event per LP, while the timestamp increment determining the actual timestamp of newly scheduled events has been set to follow the exponential distribution with mean value equal to one simulation time unit. The model lookahead has been set to a minimal value computed as the 0.5% of the average timestamp increment. Further, the overall simulation is partitioned into 4 phases where the LPs exhibit alternate behaviors in terms of updates of their states. Phases 1 and 3 are write-mild since each event only updates the classical counter of processed events and a few other statistical values within the LP state. Contrariwise, phases 2 and 4 are write-intensive, since event processing also updates an array of counters' values, still embedded with the LP state, by performing 500 updates on the array entries. Overall, the different phases mimic varying locality and memory access profiles. A classical busy-loop characterizing Phold event processing steps is also added which is set to generate an average event granularity of about 25 microseconds. In this experiment, we compared the performance of our hardware- and software-based mixed approach to both pure hardware-based reversibility (as proposed in [101]) and pure software-based one exclusively relying on undo code blocks (this is achieved by preventing any thread to exploit the HTM support in our engine). We did not compare with the performance achievable by some last generation traditional speculative PDES platform just because the data reported in [101] have shown that event granularity values of a few (tens of) microseconds do not allow this type of platforms to provide significant speedup values (due to the fact that they are based on explicit partitioning of the workload across the threads, and on explicit message passing for event cross-scheduling, thus resulting more adequate for larger grain simulation models). Overall, we assessed our proposal with a workload configuration just requiring alternative forms of speculative parallelization and reversibility support (like the one we propose), as compared to the classical ones.

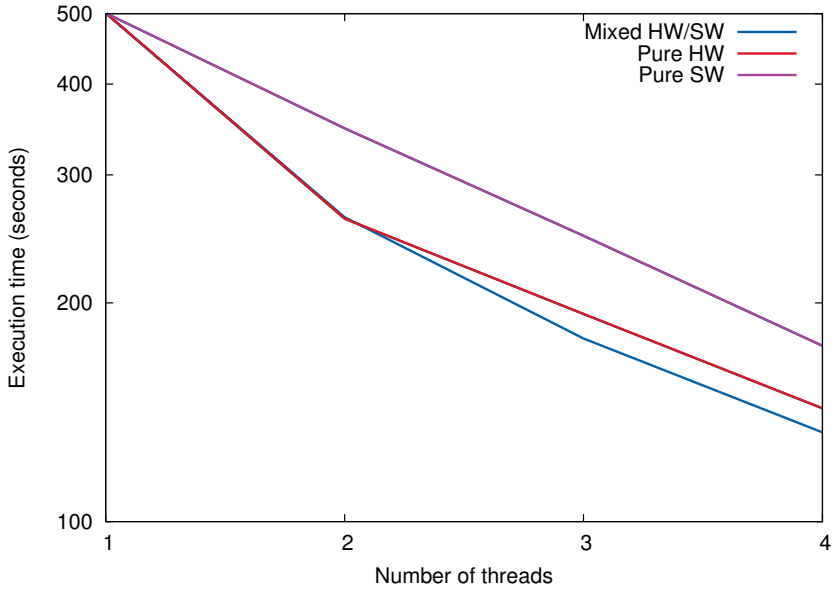
We have run this benchmark by varying the number of employed threads from 1 to the maximum number of physical CPU-cores in the underlying machine with HTM support, which is equipped with two Intel Haswell 3.5 GHz processors, 24

GB of RAM and runs Linux—kernel 3.2[6]. For the case of single-thread runs, the execution time values are those achieved by simply running the application code on top of a calendar queue scheduler.

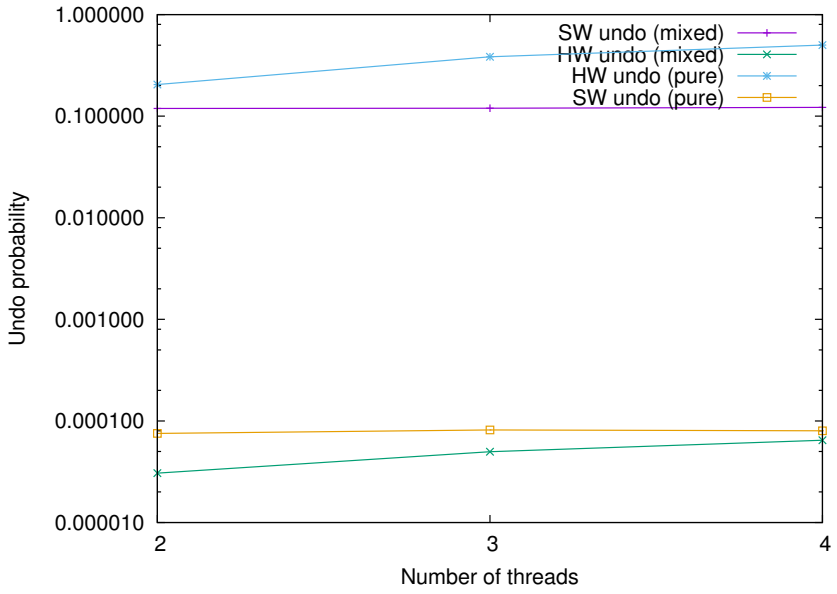
In Figure A.3a we report the observed execution time values while varying the number of threads (each reported value resulting as the average over 5 different samples). The data show how our mixed HW/SW approach outperforms both the others, with a maximum gain of up to 10% vs the pure HW approach and of 30% vs the pure SW approach (achieved when running with 4 threads). Such a gain by the mixed approach is clearly related to the fact that write-intensive phases lead the pure SW approach to become more intrusive, due to the cost of generating larger undo code blocks, which does not pay-off compared to the reliance on pure HTM-based reversibility. On the other hand, the pure HW approach does not allow the maximization of the usefulness of the carried out speculative work for larger thread counts. In fact, the slope of the execution time curve for the pure HW approach becomes slightly worse than the one of the pure SW approach when moving from 3 to 4 threads. Our mixed approach is able to get the best of the two by just avoiding excessive aborts of HTM-based transactions when relying on larger thread counts, also reducing the cost of undo code blocks' generation thanks to a fraction of events executed with HTM support. The data reported in Figure A.3b show how the pure HW approach suffers from thrashing when increasing the thread count, while the pure SW approach has minimal incidence of events undo. The mixed approach avoids the thrashing phenomenon just like the pure SW approach does, but has less overhead since executes a portion of the events via the HTM support.

Although this study is preliminary, we think that is worth reporting it as a pointer to possible optimizations of applications' reversibility supports, via the exploitation of either consolidated or innovative software based techniques—like the one we presented based on reverse windows—and emerging trends for reversibility in the hardware.

[6]The hyper-threading support offered by the processors has been excluded just to avoid cross-thread interferences—due to conflicting hyper-threads' accesses to hardware resources—which might alter the reliability of our analysis.



(a) Execution time - log scale on the y-axis.



(b) Undo probability for HW and SW speculatively processed events.

Index