SAPIENZA
UNIVERSITÀ DI ROMA

SCHOOL OF ENGINEERING

Master Thesis in

COMPUTER ENGINEERING

Application transparent and efficient mixed state-saving in speculative simulation platforms

Supervisor
Prof. Francesco Quaglia

Candidate
Davide Cingolani

Co-Supervisor
PhD. Alessandro Pellegrini

Academic Year 2013/2014

Al nostro caro Golia,
un piccolo ma inestimabile amico

Acknowledgment

However weird it may sound, the acknowledgment section may result the most challenging one to write. Indeed quite often we do not realize the everyday enrichment and gifts we continuously get from who surround us. Thereby, it will be really hard to duly acknowledge all the people that make this important objective possible.

First and foremost, I am especially grateful to my whole family, which has seen me grow but that in these few past days has seen no more than a shadow mirrored on the monitor! They has never stop to believe in me and encouraged anytime, despite my absurd answers to someone's innocent questions during the dinner time. Moreover without whose support, now, i could never be here to write on how much they gave me and how much i hope to give in future.

I owe special thanks to my supervisors, Prof. Francesco Quaglia and PhD. Alessandro Pellegrini, who made this thesis possible. They not only gave me the possibility to explore a matter of great interest, but they also have been represented a constant landmark throughout whole thesis' development.

Certainly i could not forget to acknowledge my university colleagues with whom i could share many experiences in those past years of studying, and with whom there has been a good deal of "philosophical" —euphemistically— dissertations worthy of an engineer at the coffee break! I am quite sure that the hashmap comparison will remain pretty *memorable*.

I would extend my most sincere thanks to all my friends who have always been present to me and that, together with my family, have been represented a very important point of reference. Between laughter and adventures with their affection they gave me something really invaluable. What I am now and what I hope to be in future, I owe it to them, too (Ok, jokes about what a poor job you did, guys, are pretty foregone at this point!).

Finally, a very especial greeting to Golia, a faithful and silent —mhhh, maybe not really— friend who indirectly supported me during the many moments of bewilderment with his eloquent and inimitable plush looking smile. A bond far beyond the *myth* which I hope I had been able to return with the same unconditioned passion he had always given to me with its bark, when i was home.

Ringraziamenti

Per quanto possa sembrare strano la sezione dei ringraziamenti può risultare la più difficile da scrivere perché spesso non ci rendiamo conto del contributo che ogni giorno riceviamo da chi ci sta intorno. Sarà, quindi, davvero difficile ringraziare debitamente tutti coloro che hanno reso possibile il raggiungimento di questo importante traguardo.

Vorrei cominciare con un ringraziamento speciale alla mia famiglia, che mi ha visto crescere ma che negli ultimi giorni ha visto solo un'indaffarata ombra riflessa nello schermo! Le stesse persone che non hanno mai smesso di starmi vicino e di incoraggiarmi, nonostante le mie assurde risposte alle innocenti domande di qualcuno a cena; ma sopratutto senza la quale ora non sarei di certo qui a scrivere di quanto mi abbiano dato e di quanto spero possa dare io in futuro.

Un grande ringraziamento lo devo anche ai miei relatori, Prof. Francesco Quaglia e PhD. Alessandro Pellegrini, senza i quali questa tesi non mai sarebbe stata possibile. Non solo mi hanno dato la possibilità di entrare in contatto con un argomento di grande interesse ma hanno anche rappresentato un punto di riferimento costante durante tutto lo svolgimento di questo lavoro.

Di certo non posso mancare di ringraziare anche tutti i miei colleghi di università con cui, in questi anni passati a studiare, ho potuto condividere parecchie esperienze assieme e con cui non sono certamente mancate una buona dose di "filosofiche" —per usare un eufemismo— riflessioni che solo un ingegnere alla pausa caffè può trovarsi ad intraprendere! Sono sicuro che nonostante alcuni timori rimarrà *memorabile* il paragone dell'hashmap.

Ancora, un grazie davvero sincero e pieno d'affetto lo rivolgo ai tutti i miei amici che sono sempre stati presenti e che assieme alla mia famiglia hanno rappresentato un importante punto di riferimento. Tra risate ed avventure con il loro affetto mi hanno regalato qualcosa di davvero inestimabile. Quello che sono e che mi auguro di poter diventare, lo devo anche a loro (Andiamo, battute su quale pessimo lavoro abbiate fatto sono scontate a questo punto!).

Infine, un particolare saluto a Golia, un fedele e silenzioso —si fa per dire— amico che mi ha indirettamente sostenuto durante i tanti momenti di smarrimento con la sua tanto eloquente quanto inimitabile espressione da peluche. Un legame che va ben oltre il *mito* e che spero di aver saputo corrispondere con lo stesso affetto che incondizionatamente mi ha sempre regalato con il suo abbaio, al rientro a casa.

Contents

List of Figures

List of Tables

Introduction

Rollback process is unavoidable in many complex systems, from parallel to distributed applications, to prevent system crashes and to undo portions of speculative execution belonging to an inconsistent trajectory or misbehaving operations. The most consolidated way to realign the system to a consistent state is to undo part of the computation which brought system to a generic misbehavior. In database and distributed systems, colliding elements are typically transactions trying to access shared resources simultaneously, but even in much simpler architectures conflicts at instruction level can occur, i.e. speculative executions or branch predictions. Like other systems, parallel applications introduce a high contention factor on shared resources. Optimistic parallel simulation applications, process millions of events speculatively, many of which might be rolled back due to relaxed synchronization constraints. The largely adopted solution, for the all the aforementioned cases, is to rollback whole system to a previous coherent saved state; name the state saving technique.

Generally, consistency and fault-tolerance are fundamental properties that parallel or distributed systems must exhibit to ensure reliability. Analogously, parallel applications heavily exploit available resources, increasing the likelihood to lead into out-of-time states. At the same time, concurrency among generic memory transactions and nodes' processes are subject to failures which hurt the whole system causing it to lay into an incoherent state. In simulation platforms, rollback is due to straggler events that overcame speculatively yet processed events. If one operations causes system to misbehave, it leaves inconsistent data behind that have to be fixed in some way. The naïve solution to *cold restart* from the very initial state, is clearly unfeasible in complex application due to exceeding time requirement, beside the computational waste. To ensure a minimum performance level, systems must be rolled back to the nearest stored consistent state's snapshot, which requires to maintain a program history structure. Upon a misbehavior, the framework rolls back to the nearest checkpoint in time, loosing a restrained portion of computation. Once the system has been restored, forward execution flow is re-established. So far, the simple and fairly expensive record&replay mechanism was employed, but what if the system would be able to act "in reverse"? Contrarily to checkpointing, we could adopt a backward approach which realigns the whole system to a consistent state by simply rewinding computational steps, instead of restoring a costly program snapshot.

Reversibility, on the other hand, is a very appealing alternative to classical state-saving rollback. Despite one can imagine, reversibility is a quite ancient research field. However only in latest years it has gained a growing attention, both for energy efficiency aspects and for software performance. It may have several potential applications in computer security, transaction processing, intrusion detection systems, debugging activities, developing aiding tools (or IDE[1]), and backtracking reasoning.

In this thesis we focus on devising a reverse code generator and on its integration with speculative simulation platforms, and specifically for Parallel Discrete Event Simulation (PDES). Speculative simulation platforms usually require a considerable storage allocation to model the reality and to store partial results, which further experience several update per time; nonetheless, the optimistic slant increases the likelihood of rollbacks. Simulation processes exhibit CPU-intensive burst loads too, representing a perfect field of application to test our approach in all the possible cases at once. Simulation is a problem-solving technique to cope with complex mathematical

[1] Integrated Developing Editors

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 relies on the *virtual time* concept, processing the incoming messages. The event-based simulation enforces equivalency between those messages 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 *safe*. Event safety straightway depends on actual processing events order with respect to global causality relationship they have been sent with. Optimistic approach looses event processing constraints and exploits much better computational resources, nevertheless it might bring the system to violate the causal order, bringing the simulation to an inconsistent state. The more complex simulation model is, the more likely it requires to rollback out-of-order events. So far the most consolidated way is to employ checkpointing techniques, which though exhibit considerable memory overhead and time latency as the simulation model gets more complex.

This thesis specifically aims at realizing an efficient reverse code generator module for speculative simulation application within the High Performance Computing (HPC). We propose a novel approach based on a hybrid strategy that interweaves classical state-saving technique with reverse computing, conceived to reduce as much as possible computational and memory requirements. Present work's main goal is to give the most comprehensive perspective of our proposal, from high-level intuition deep into implementation details and performance assessments. Chapter 4 "The reversibility architecture" dives into a thoroughly dissertation on how reversibility is achieved through this mixed reversible state-saving technique, and how it integrates into the simulation framework we adopted (Section 4.5 "The use case: ROOT-Sim"). As hinted above, simulation requires a huge amount of memory which is, furthermore, intensively updated. State saving could lead into exceeding memory overhead tackled by widening the checkpointing time interval. This a quite solid working solution which, though, increases time effort to realign whole system to a common virtual time. On the contrary, the reversible program's history would be basically constituted by the reverse of each executed instruction, allowing to go back to any prior point in time just by re-execute them. No snapshot is required, and less overhead data structure complexity for maintaining state attributes, from which follows a reduced time effort. According to the instrumentation method, reverse code is statically predefined or dynamically generated when needed. Nevertheless, in both cases it becomes part of the original code seamlessly.

The major challenge is to determine which instruction to reverse and how to compute its inverse. We minimized the impact of latter factor at the expense of initial static pre-processing, such to guarantee to not hurt runtime performance dramatically. We achieve reversibility through a hybrid instrumentation strategy. We intentionally chose to instrument relocatable objects via a static and rule-driven approach. According to user-provided rules (via simple *xml* configuration file), input code is persistently enhanced by injecting well-structured code blocks, devised to emit reverse executable code *on-the-fly*. Commonly, there are two main ways to realize instrumentation, either *statically* or *dynamically*. Independently to which method is used (Section 1.4.1 "Ways to instrument the code"), the introduced overhead is mainly composed of (a) a bigger binary file due to the injected code, (b) the time required by the system to execute extra code, and (c) an additional memory amount to store possibly dynamically generated code. To understand this third element, it sufficient to consider instrumented self-modifying code which might enlarge original binary at runtime. Indeed this is what our reversing module does. The reason behind static instrumentation choice is to limit time overhead due to dynamically parse and interpret ready-to-execute instructions to decide whether to reverse them or not. Unlike the dynamic approach, it does not leads into an exceedingly time-consuming process, which would affect program's efficiency. Nevertheless, the static paradigm marginally introduces some challenges and reduces tool flexibility (see Section 1.4 "Instrumentation").

One of the big hurdles of common static instrumentation tools is that there is no simple way

to infer software's semantic which requiring a notable complexity in its handling. To understand why the semantic is so fundamental, it is sufficient to bear in mind that unlikely machine instructions and logical operation are in a one-to-one mapping. The assembly code is a low-level language, optimized by compilers for specific machine architecture. It does not provide any information on logical operations, nor about which instructions they involve, thereby it is not possible to straightway reverse assembly code ignoring the underneath semantic. To overcome aforementioned hurdle, we picked advantages from both strategies. Reverse computation's strength, on the one hand, and state-saving simplicity on the other. We focuses to reverse only those instructions which directly alter memory locations. In principle seemingly to what the incremental state-saving technique does, instead of storing the whole state in a separate storage unit, our module converts it in a reversed set of operations, generated on-the-fly by the runtime support. Instructions execute faster with regard to any snapshot restoration process. Nevertheless, we aim to best optimize resources, as well. However big the container might be, many drops eventually fulfill it, so even instructions have to be restrained. The reverse code generator focuses on memory-write instructions only, which are reversed and directly stored into the program's heap. In this way, we create a dynamically growing module which provides to the final software, the entry points in order to be self-reversible. Simulation program itself —or rather any other overlay software in its turn, for a future interoperability— will rely on this support to dynamically invoke or step the reversed functions, as needed. From an high-level perspective, inverse functions get seamlessly part of the software, and therefore can be called as generic functions without means of interrupts or wrappers.

The instrumentation process is transparent to the user, the system autonomously retrieve input format and machine specifications, and remaps binary code into a more convenient representation. We have been adopted an *ad-hoc* binary representation to decouple the input details from the instrumented process. Since its modular design, this project is intended to be as extensible and flexible as possible, able to support any combination of executable file format and machine instruction sets.

The efficiency of this approach is evaluated by supporting two practical experiments, discussed further in Chapter 5 "Experimental Assessment". As benchmark we chose a personal communication system simulation model which runs on the open source ROOT-Sim [**?**] platform (Section 4.5 "The use case: ROOT-Sim"). We are foremost steered to unearth the bare overhead introduced by our approach, giving a baseline efficiency; further, a major emphasis is given to assess system's performance in a real simulation process. We believe this approach would reduce time effort providing a valuable speedup, if compared to state-saving mechanism, as the simulation complexity increases.

Beyond the scope of the present work, we are likewise interested in exploring possible future applications of our module, e.g. debugging scenarios. This thesis paves the way to the development of a comprehensive framework that provides necessary prerequisites to properly support reversible computation for a range of applications. Even though this module is not straightway related to program analysis purposes, it may pave the way to a new form of debugging support, i.e. the reversible post-mortem debugging (Chapter 6 "Future application fields"). Future work consists in extending this strategy to a more general debugging aspects, instead of implementing it in a single-purpose environment, exploring how to revive a crashed program and its whole environmental context.

The remainder of this thesis is structured as follows: Chapter 1 deals with reverse computation theory, giving a comprehensive perspective on the context within which this thesis resides. Further, it analyzes rollback techniques, compared to reverse execution capabilities, to finally describes instrumentation techniques, to better understand how it is used in the present work to achieve reversibility. Chapter 2 gives an overview on related work on debugging and instrumentation implementations. It highlight peculiarities of each relevant implementation encountered during our research, highlighting features, difficulties and the way they solved them. Following,

Chapter 3 discusses the instrumentation framework's architecture —namely Hijacker—, from the high-level viewpoint down to the inner implementation aspects. A particular attention is focused on decisions to implement the reverse code generator module we devised. For the sake of clarity, the chapter gives a description on the configuration files driving the reversing module. Chapter 4 describes reversibility by means of the Hijacker's instrumentation module. The chapter is a detailed dissertation on our specific approach, unearthing challenges and solutions, along with a reference to the introduced overhead. It deals with implementation aspects of the reverse code generator from an higher point of view to a more detailed one. Specifically it represents an insight on our hybrid reversible approach which realizes a mixed state-saving technique. To provide a thoroughly perspective of performance assessment's context, Section 4.4 covers a detailed description on parallel optimistic simulation environments. It further faces adopted solutions in simulation landscape, relating them, on the other hand, to what reverse computation allows to achieve. Finally, in the Chapter 5 we discuss experimental results achieved. The chapter describes experiments' structure and objectives, which are to not only insight the pure time and storage overhead introduced by our module, but the influence on a real simulation process. Additionally, Chapter 6 is dedicated to discuss about future employments of our approach, specifically towards reversible debugging scenario.

CHAPTER 1

Reversibility details

This chapter is intended to give an insight on the playground on which this project relies. The main project's goal, as mentioned in "Introduction", is to focus on providing the necessary to support hybrid checkpointing system pursued through an advanced instrumentation approach. This project, indeed, paves the way to build a modular enhancing architecture spanning over different fields, from the improvement of debugging and profiling systems, to efficient rollback paradigms by employing reversible computing techniques. Scope of this thesis is to describe and evaluate an innovative hybrid approach to solve state-saving rollback approach by means of reverse computation. Though, our framework is fundamental an instrumentation tool, which provides a set of modules targeting specific problem instances. Thereby this chapter aims to thoroughly unwind instrumentation techniques proposed so far, but first it will explore the arcane beyond reverse computation and the aforesaid state-saving paradigm. It is a crossroads to conceive how Hijacker interweaves those different techniques and paradigms keeping the advantages from each one in order to achieve a higher global efficiency. Although the project is, actually, still targeted to the undertaken case of study, we would like to depict project's possible future capabilities by providing such an overview spanning over the evolution history of this research field.

Reversible computation provides valuable opportunities in a wide field of applications which spans from database systems to bidirectional debuggers, and from intrusion detection systems up to massive parallel and distributed systems. More generally, it applies to whatsoever system which requires to undo erroneous or unwanted operations. The prevailing adopted solution to provide rollback support is, actually, the widely affirmed state-saving based approach, though it could be quite expensive. On the contrary, reverse computation could provides, in theory, an interesting performance speedup. Nevertheless, state-saving much simplifies the architectural aspects with respect to reversibility strategy; thereby, herein the purpose is to overcome common hurdles and expenditures, which rollback-based systems exhibit, via the reverse computation approach.

1.1 Reversibility

We intentionally open this section by giving the definition of what the reverse computation allow to *do*, instead of what it *represents* in fact. This, to foremost give the concept beneath its potentialities in order to easily understand how will be achieved.

Definition 1 (Reversibility)**.** *Any program is defined to be reversible if it allows to rebuild any prior system's state relying on the only knowledge of the current one by backward retracing all the computational steps between them. In other words, reversibility implies that each state S_k has a unique predecessor S_{k-1}.*

Reverse execution is in fact the software implementation of the more general theory of *Reversible Computing* which, in turn, refers to computational models performing only *logically*

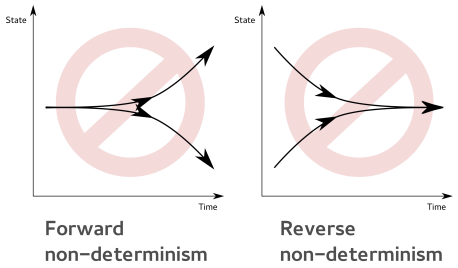


FIGURE 1.1: *Divergent computations*

Bit value	0	0	0	0	0	···	0	···	1	1	1	1	1	···	1
Entrpy	1	2	3	4	5	···	N		1	2	3	4	5	···	N

Irreversible Computation

Bit value	0	0	0	0	0	···	0	···	1	1	1	1	1	···	1
Entrpy	1	2	3	4	5	···	N		N+1	N+2	N+3	N+4	N+5	···	2N

FIGURE 1.2: *Entropy variation in bit-erasure computation*

reversible operations. That is, instructions which allow to rebuild any previous state relying on the only knowledge of the current one, proceeding backwards through the computational steps performed. Fundamental requisites to achieve real software and physical (§1.2) reversibility are the following two: (*a*) no information have to be lost or implicitly ignored during computation and (*b*) the transition function, from a generic state S_n to state S_k, must be bijective. Nevertheless, there is no existing software implementation designed to be fully reversible, actually they always exhibit a certain degree of irreversiblity as leading inheritance of the programming languages' structure. One of the biggest hurdles in achieving software reversibility is indeed due to the forward-determinism of our traditional computational models and, thereby, of their relative programming languages; however some overcoming solutions were proposed, in the last decades (§1.2.3).

Definition 2 (Logical reversibility)**.** *An operation is logically reversible whether its logical state just before it is performed, is uniquely determined by the output sate; thereby, no information must be erased, or otherwise lost. The straightway consequence is that a logical reversible program is backward deterministic along its timeline.*

Definition 3 (Physical reversibility)**.** *Processes are defined to be physical reversible if they do not dissipate energy in heat at all, hence without generating entropy. Though, perfectly physical reversible machines are unfeasible, as the quantum theory clearly explains. Any environment has always a non-zero possibility to be irreversibly altered by an everlasting ground noise which creates entropy.*

1.2 Physical reversibility

Although in the past it did not represent a major hurdle, nowadays power consumption gains an increasing attention since the growing number of embedded and mobile devices featured by a limited power supply and heat dissipation means. Therefore, besides the due respect to our Planet, managing wastes with a much more careful perspective has became mandatory. Relying on this aspect, aforementioned *Reversible Computing* theory is hence an interesting field of research since it would allow to develop more efficient, but even powerful, hardware devices. It does not represent a mere high-level implementation within software, programming languages or algorithms, but it also straightway pursues *physical* reversibility. A quite wide research area born in the '60, is targeted at developing physical reversible devices which are able to perform invertible operations; transformations carried out by physical mechanism that are (almost) thermodynamically reversible, or rather which implements a zero-entropy operating logic. Needless to say that it's quite impossible to develop a perfectly reversible hardware devices in reality. However the continuous evolving of this model will eventually land, and in fact it already does, to very interesting practical results.Physical reversibility is a notable topic, but it lies out of this thesis' scope and even though software and hardware reversibility are deeply related to each other (as Michael P. Frank shown in [**?**]), this chapter will only focus on the former kind of model. Nevertheless this section is dedicated to give the reader an overview about this topic, in order to keep not hanging up those concepts. For a deeper insight on this matter, refer to [**?**, **?**, **?**].

$$\textit{Logical reversibility} \iff \textit{Physical reversibility} \tag{1.1}$$

Unfortunately, traditional models of computation are based on logically *irreversible* schemes which entail the irreversibility property down to the machine instructions. Although reversible hardware devices still remain more prototypes rather than real implementations, several endeavors were made. Attempts to achieve reversibility in classical architectures could led into the frustrating result of decreasing the overall computational efficiency. However, as P. Frank claims in [**?**], traditional models ignores fundamental physical constraints which can lever on to obtain a valuable gain. We quote from the abstract of his work:

> [...] This thesis gives the first analysis demonstrating that in a realistic model of computation that accounts for thermodynamic issues, as well as other physical constraints, the judicious use of reversible computing can strictly increase asymptotic computational efficiency, as machine sizes increase. I project real benefits for supercomputing at a large (but achievable) scale in the fairly near term. And with proposed future computing technologies, I show that reversibility will benefit computing at all scales. Next, the thesis demonstrates that reversible computing techniques do not make computer design much more difficult. [...]

Conservative logic

Logical reversibility of whatsoever program is indeed strictly related to the underneath *physical* reversibility as well. Traditional hardware models are actually irreversible since they dispose information by performing computations. Relying on this forward-deterministic hardware architecture, it is not surprising that also high-level languages are irreversible in nature, since they are developed over such a kind of architecture. Landauer's principle [**?**], states the relation of the heat dissipation with regard to the bit information erasure; he also argues the inevitable presence of logic (thus physical) irreversible operations in traditional computational models.

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

$$E_{min} = k_B T_a \ln 2 \tag{1.2}$$

Where the 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.

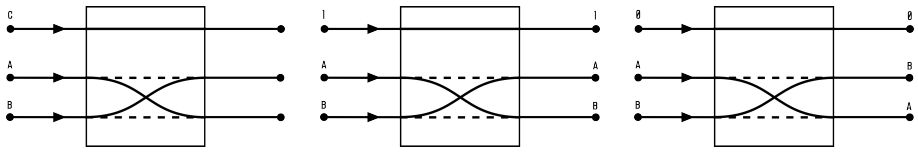
E_{min} is the corresponding amount of energy that any computer will at least dissipate as heat for each single bit of information erased, or otherwise disposed[1]. Going against Landauer's claim, Toffoli in '80 realizes [?] a working hardware prototype of physical reversible logic gate. Subsequently, Fredkin et al. in 1973 [?] investigated the possibility to build a complex reversible and also *conservative* hardware architecture, by the employment of the Fredkin logic gate. A conservative-logic gate, such as the Fredkin gate, is a Boolean function which is either invertible and conservative. Former property regards the mathematical aspect to be able to find an invertible function, whereas the latter refers to the real information conservation throughout the elaboration process.

Definition 4 (Reversibility)**.** *Let A be an automaton, it is reversible if the transition function from generic state S_k to generic state S_n is invertible.*

Definition 5 (Conservative)**.** *Let N be a combinatorial network, it is defined to be conservative if it conserves in the output the number of the `0`'s and `1`'s present in input.*

C	A	B	$\hat{C}$	$\hat{A}$	$\hat{B}$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

TABLE 1.1: *Fredkin's gate truth table*



(A) *Symbol* (B) *Operating logic*

FIGURE 1.3: *Fredkin gate logic*

Conservative logic considers processing merely as a signal stream conditional routed within the network. To create a parallelism with the quantum mechanic theory, matter cannot be created nor destroyed, only transformed.

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

1.2.1 Logical reversibility

Basically, the irreversibility property of current platforms is due to the destructive logic on which they rely. Most often information is disposed throughout the operation. To quote from [**?**]:

> Currently, computations are commonly irreversible, even though the physical devices that execute them are fundamentally reversible. At the basic level, however,matter is governed by classical mechanics and quantum mechanics, which are reversible. This contrast is only possible at the cost of efficiency loss by generating thermal entropy in tot he environment. With computational device technology rapidly approaching the elementary particle level it has been argued many times that this effect gains insignificance to the extent that efficient operation (or operation at all) of future computers requires them to be reversible.

Branches, assignments or other non-deterministic functions (e.g. random number generators) represent a subset of the instructions that implicitly gobble bits of information. In the reversible computing theory those instructions are referred to as *destructive operations* —Table 1.2 reports the most common operations. Besides entropy that those operation generates by flipping bits' state, they basically prevent to rebuilt prior states by a backward retrace. For example, an `if` statement is solved during forward execution basing on the conditional input value; however that value will be just thrown away as soon as the proper branch is taken. On the contrary, backtracing the `if` statement would require to keep the information of which branch has been taken. Considering the following code snippets (in `C`-like syntax).

```
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 1.1: *Indeterimism of code branch implies irreversibility*

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

LISTING 1.2: *Reversible implementation of the code branch*

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

LISTING 1.3: *Inverse function that undoes `foo`*

Listing 1.1 clearly shows that it is impossible to know from which of the two branches the statement at line 10 is reached, since this information will be definitively disposed. Once the branch is taken, there is no longer the information suggesting whether was `value` to be less than `offset` or rather the opposite. Therefore there is no possibility to trace backwards the execution from line 10 but in the case we have knowledge of which branch has to be rewound. On the contrary, Listing 1.2, depicts a reversible and semantically equivalent[2] version of the previous code snippet. In this case the `b` variable retains the output of the comparison, allowing to properly undo operations, which are performed by the relative reversing function shown in Listing 1.3.

Programming languages' statements can be categorized in two general families, (*a*) *constructive* and (*b*) *destructive* operations. The constructive operation are those in which no information are lost, contrariwise destructive ones overwrite one or more operands making impossible to trace them back. Table 1.2 shows a rough categorization of the most common `C` operands, along with their possible inverses. Due to the traditional programming languages constructs' nature, destructive operations are eventually employed in any program. Integer division, shift and modulo

Operator	Statement	Meaning	Type	Inverse
=	`a = b`		destructive	`SAVE(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	`SAVE(a)`
%=	`a %= b`	`a = a % b`	destructive	`SAVE(a)`
≪=, ≫=	`a ≫= b`	`a = a ≫ b`	destructive	`SAVE(a)`
|=	`a |= b`	`a = a | b`	destructive	`SAVE(a)`
&=	`a &= b`	`a = a & b`	destructive	`SAVE(a)`
++	`a++, ++a`	`a = a + 1`	constructive	`a--`
−−	`a--, --a`	`a = a - 1`	constructive	`a++`

TABLE 1.2: *Assignment statement constructiveness*

operations, in general cause loss of information for arbitrary variable values. Integer division may cause truncation of 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.

Several solutions have been proposed so far in the literature to address the logical reversibility problem (§1.2.3), spanning over several abstraction levels and approaches.

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

Instruction-level reversibility is the finest one, theoretically allowing to walk back and forth along the whole execution timeline (except for the aforementioned destructive instructions). A

[2]Research of reversible languages and semantically equivalent is a way towards reversibility, however it requires to "understand" the code itself, namely infer its semantic.

perfect instruction-level reversibility represent a kind of optimum in terms of time and memory requirements, since every forward set of instructions may be undone by the relative inverse. Therefore, no variable's trace must be kept and no extra instructions would be required, hence forward program's execution would not be slowed down. However, this perfect instruction-level reversibility is quite impossible to obtain due to the aforementioned structure of our traditional computational models that inevitably rely on destructive operations (or instructions). Feasible solutions unavoidably stain the purely reversion process by overcoming the information erasure with some form of state saving, thus introducing a non-zero overhead. At a first glance, taking into account aforementioned considerations, it might seem that a similar approach is definitively not worth. In fact, from a theoretical perspective, rewinding the computation by overturn instructions allows to better exploit computational resources. Herein we anticipate the Akgul's work [**?**], where he devised a instruction-level reversion algorithm, by quoting the achieved results. To understand the quotation, ISS stands for *Incremental State Saving* and ISSDI is *Incremental State Saving for Destructive Instructions*, whereas the RCG is the reversion technique algorithm adopted:

> In order to test instruction level reverse execution on a debugging session, we implemented a low-level debugger tool with a graphical user interface (GUI) which provides debugging capabilities such as breakpoint insertion, single stepping, register display and memory display. The debugger runs on a PC with Windows 2000. The PC is connected to the MBX860 board via a Background Debug Mode (BDM) interface. [...] However, even in this case, the runtime memory requirement with RCG is 82X and 55X smaller than the runtime memory requirements with ISS and ISSDI, respectively. In general, RCG achieves from 2.5X to 2206X and from 2X to 1404X reduction in runtime memory usage as compared to ISS and ISSDI, respectively. [...] The slow down in reverse execution with RCG as compared to ISS and ISSDI is between 1.16X and 1.89X.

Indeed, only a reduced subset of instructions are actually undone leaving the remainder unaltered, but moreover it requires much less storage either to save and to retrieve values. Memory access is quite expensive with regards to instruction execution ($\sim$3 orders of magnitude), therefore along with the memory thrift, it follows a time overhead saving. Reverse history has only to keep track of a very reduced amount of information, such as the branch-path, and allows to optimize the runtime slowdown. However reverse computation introduces other thorny issues concerning common non-deterministic interactions or functions (e.g. random number generator) which require to be rerun within the same first execution context.

1.2.2 Feasibility of the reversible execution

Usually digital computers perform operations that dispose information on the computing history. Therefore the immediate predecessor of a computational state could be ambiguous. In other words traditional general-purpose automaton lacks a single-valued inverse function.

In 1973 Charles Bennett [**?**], first showed the theoretical feasibility of logically reversible computers, arguing that in fact 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 in real and complex hardware architectures. However, his work was hinged to overcome Landauer's claim [**?**] that irreversibility is an unavoidable computing property. Bennett built a 3-tapes reversible Turing Machine which proved the reversibility of general-purpose computers, which is not much more complicated than irreversible ones. In fact, this is the first attempt in this direction. However Bennett's reversible machine is based on the *state saving* technique; basically he saved all the information that would be otherwise disposed. Evidently, such a machine may require a very large amount of storage, fact which

renders a real implementation in complex systems quite unfeasible.

In 2001 Buhrman et al. in [?] proved time adn space upper bound to reversibly simulate irreversible computations:

> 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.

1.2.3 Approaches to reversibility

Program's reversibility is a wide and general field which embraces lots of subtle viewpoints, form logical to physical ones, such as the realization of hardware architectures able to perform naively reversible execution. So far, Definition 1 of reversibility introduces only a conceptual perspective. Reversible execution (or computation) is indeed achievable also on top of traditional irreversible machine. Hence, how reversibility is practically achieved?

The obvious and foremost plain solutions is the *state-saving*. As the name suggests, program's history is periodically stored as checkpoints during the natural execution. Therefore as soon as the system is prompted, any previous state can be reached by restoring the process state recorded in the snapshot. Since checkpoints are discrete in time, though, this leads into a coarse granularity. To restore a previous sate it has to be chosen the nearest checkpoint in time and then replay those instructions which separates the checkpoint from the target time. Although this solution is quite simple and requires low architectural complexity, it is highly time-consuming and moreover it may require a very large amount of storage. Further a bounce of optimization were made to improve the state-saving strategy leading to the *incremental state-saving*. In the incremental state-saving, only the directly modified states are stored, therefore it slightly lightens the memory requirements which, however, remains still considerable in complex systems, such as parallel and distributed application. Further, unlike transactional applications, in general purpose computation or in parallel simulation it is harder to recognize variable modifications, making more complex the design of the state saver.

Program animation is another approach to reversible execution. It basically interposes a virtual machine level with a reduced and reversible instruction set. Therefore each real assembly instruction is uniquely mapped into a reversible one allowing to run it backwards. Since the program must be dynamically interpreted, 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. For those instruction state saving is applied. Even though, time and memory requirements are lightened, they still slow down the forward normal execution since other extra instruction have to be performed.

A very interesting solution is the one proposed by Tankut Akgul and his teamwork in 2002 [?]. They proposed a novel approach which acts at the assembly level. 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. Since no additional instruction has to be executed, this solution would theoretically introduce almost-zero time overhead, except the time to rewind calculi; further it requires much

less storage amount. A portion of memory is mandatory due to the aforementioned destructive operations. Whenever the algorithm cannot to straightway reverse the assembly code, it will fallback to state saving. Nevertheless, this kind of solution is also quite complex to realize and to implement. The major hurdle in this way is to infer software logic. The semantic is mandatory to known how to group assembly instructions that belong to the same logical operation. Akgul relies on dynamic control flow information from which he builds a Control Flow Graph (CFG) employed to properly reverse instructions blocks. 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 SSA.

Another field of research in the direction of the reversibility hinges on devising reversible programming languages. However, the major hurdle is that the writing of a reversible software in nature overburdens the developer with an extra effort. It is difficult to achieve due to the non-conventional concealing and the is unreasonably time-consuming. Rather, it would be much more valuable to develop a system able to convert any irreversible program into a reversible one.

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 [**?**], face the principle that a programming language (independently of the abstraction level) must to ensure in order to be reversible, further they devise a reversible high-level language prototype. Noteworthy it is what they claim:

> [...] under the assumption that inverse constructs have the same computational complexity as the forward ones, a language will be easily and locally invertible, and the inverted programs will have the same complexity as the forward programs. [...]

However, devising a program directly in a reversible flavor is not a straightforward process. This is why reversible languages, though feasible, are not really employed. It would be much more convenient to circumvent the problem by using a proper interpreter which generates the relative forward and reverse version of the program. As again [**?**] states:

> Currently, almost no algorithms and other programs are designed according to reversible principles (and in fact, most tasks like computing Boolean functions are inherently irreversible). To write reversible programs by hand is unnatural and difficult. The natural way is to compile irreversible programs to reversible ones. This raises the question about efficiency of general reversible simulation of irreversible computation.

However also this strategy poses the challenge of whether to instrument the native code or to dynamically interpret it (§1.4.1).

1.3 The Rollback Operation

Rollback is required by any environment in which a prior state should be restored, to prevent system crashes due to failures, to undo speculative steps which are subsequently considered inconsistent, or in general to rewind erroneous or unwanted operations. Many systems exhibit a certain level of concurrency; therefore, they need a rollback engine to solve the unavoidable conflicts among those operations. The most effective way to realign the system to a consistent state is to undo the part of the computation which brought the system to the faulty state. In database and distributed systems, the colliding elements are typically transactions tying to access

shared resources simultaneously, but even in much simpler architectures conflicts at instruction level can occur, i.e speculative executions or branch predictions. Parallel optimistic simulation platforms, process hundreds (or even millions) of events speculatively, most of which might be rolled back due to the relaxed synchronization constraints. A solution to cope with all of these challenges is to rollback the whole system's state to a previous coherent one. Commonly, rollback support is actually achieved through state saving techniques.

Definition 6 (State saving)**.** *State saving (or checkpointing) is the act of saving the relevant state of a running program so that it can be lately restored.*

State saving provides the backbone on which rely a great number of software tools and system architectures. Database systems, fault-tolerant distributed systems, speculative techniques in parallel application and microprocessors architectures. A middleware is usually in charge of periodically taking a snapshot of the current system's state before memory transactions[3] take place and traces any relevant operations belonging to them. Therefore creating a program's history maintaining persistent and volatile altered data. Although this kind of strategy allows to provide a relatively transparent rollback support [**?**], it exhibits burdensome limitations. For quite complex and long executions, the storage requirement would grow unreasonably due to possible frequent state changes and memory saturation; further each memory access to store or retrieve the data will increase the overall number of instructions to be performed also in the forward flow. Thus, state saving could heavily affects program's performance. Although this strategy has evolved experiencing various improvements, several implementations focus and optimize only application-dependent issues.

Prior to rollback is necessary keep track in some way of the program's evolution, namely its history. That program's history is basically composed of a list of recovery records, roughly classified as *state-* or *operation- based*, according to the kind of information they provide.

state-based → ***physical logging***
Represents the physical memory record of one variable's state or register value, along with possible location in memory of that value.

operation-based → ***logical logging***
Describes the operation which alters some variable rather than a memory state.

Many implementations of checkpointing engines have been developed and improved, aimed to reduce overhead. A deeper insight on classical checkpoints and their implementation can be found in [**?**]. A dynamic approach to checkpointing which does not require to instrument or wrap any library is given in [**?**, **?**] with the DyMeLoR and Di-DyMeLoR[4] software, respectively. They are software layer designed to minimize memory consumption due to meta-data describing the current layout of the simulation process's state, and to provide an efficient and transparent facility to allow simulation objects scattering across non-contiguous memory chunks.

Narrow as it may be the gap between two successive checkpoints, the system always needs to be realigned to the precise faulty point by a forward re-execution from the snapshot just restored, namely the *coasting forward* process. In database systems, for example, when a transaction has to be aborted, a rollback takes place. The rollback manager would find the nearest checkpoint and rolls back the whole system to the previous consistent state, afterwards it redoes transitions up to the current point ignoring the faulty one. Analogously, in simulation systems, state is then reconstructed by the message log stored along with the checkpoint throwing away faulty branch computation steps.

Generally, rollback results in a twofold time effort. First it slows down the execution speed in order to update the program's history, and second it introduces a computational waste due to the

[3] In this context we refer to *memory transactions* as a generic state transitions which involve one or more variables' change (i.e. memory region, register, stack, heap).

[4] DyMeLoR stands for *Dynamic Memory Logger and Restorer*, and Di-DyMeLoR refers to *Dirty-DyMeLoR*.

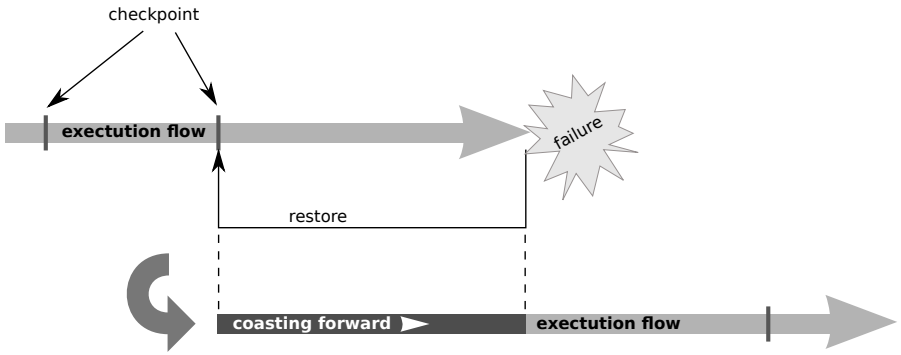


FIGURE 1.4: *General rollback scheme*

coasting forward. Considering that in database systems, distributed or simulation environments rollback is employed to realign the system back to even one faulty generic transaction, expense becomes even more evident. In those cases, it would be sufficient to undo only the operations related to that transaction, rather than to costly find and restore a previous state which likely requires a partial replay. Overall, the waste is not only energetic but moreover in performance due to the time required to realign the system and, from an higher perspective, in algorithm complexity to maintain consistency in a distributed (or parallel) environment.

1.3.1 State saving

During a program execution, its state will change several times due to the operations performed. As the name suggests, through sate saving each time an instruction alters the program state, a generic structure containing all the relevant information will be saved. That structure is basically represented by(a) the set of registers, (b) the set of global variables, (c) the current stack and finally (d) the actual heap. The state-saving technique may require a considerable

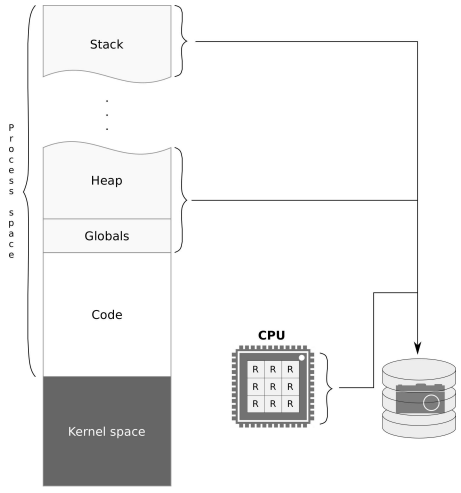


FIGURE 1.5: *The process' history snapshot elements*

amount of memory to store such information, proportionally to the execution length and the

memory image to be checkpointed. Therefore reversibility is basically bounded by the actual system storage capabilities. In parallel and/or distributed systems, the overhead introduced could be unreasonably high, so as to prevent an efficient and complete state restoring. For example, instructions beyond the threshold due to system's memory boundary, cannot be restored anymore. To overcome storage boundary, a wraparound buffer solution may be employed; however in this case an additional time cost required to reclaim old memory already enrolled by old state snapshot must be taken into account. Thus, the overhead is mainly constituted by (a) the storage requirement to store the process' state, (b) the time cost to take the snapshot, which introduces a delay between the natural execution and the checkpoint creation, and (c) finally, the cost of communication and synchronization among processes to take consistent snapshots and. Let T_k be the timestamp of last saved checkopint. It follows that restoring the systems to the generic time t, such that $t > T_k$, requires to look for the nearest checkpoint T_k, to restore it, and than re-executing each computational step until the wanted state at time t is reached.

Checkpoints must be thoroughly consistent within multiple nodes of the systems, and must be complete with respect to the restoration process. A big effort is required to create and maintain those snapshots, so that it may hurt system's performance considerably. In order to keep checkpoints consistent, a relatively high number of information is required. In distributed systems, preserving consistency implies coordination among nodes throughout the snapshot process and requires message causality to be enforced. However, since rollback are not so frequent, the failure-free running overhead is considerable. Parallel simulation applications, on the other hand, employ the state saving to ensure event causality. A quite interesting research on state saving techniques applied to PDES platforms are presented in [**?**].

A checkpoint is logically a memory snapshot (Figure 1.5). Relevant state information is collected in a log which represents the whole program's history. An optimization is to save only altered values, rather than the whole set, namely the *Incremental State Saving*. Therefore, instead of saving the program's state each time one operation modifies it, this approach takes a differential snapshot of the incremental change set. In such a way memory consumption is reduced, since checkpoints are discrete in time; nevertheless rollback requires to replay each step forward form the previous checkpoint until the desired program point is reached.

Generally checkpointing process should be transparent with regards to the application's programmer. He/she likely ignores the underneath state saving aspects, such as consistency issues. Therefore it risks to overburden the developer with the devising of a complex architecture leading to misleading checkpoints, resource wastes or dangerous behaviors. In the following, we provide a very brief overview of the most common state saving techniques.

Copy state saving Copy State Saving (CSS) is the simplest technique firstly proposed by Jefferson in [**?**]. Basically it consists in a plain copy of the whole program's state and possibly relative meta-data to restore it. Whenever an operation affects the state, a snapshot is taken.

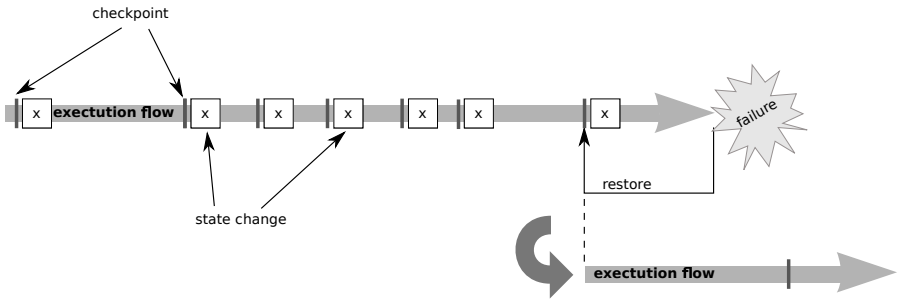


FIGURE 1.6: *State-saving rollback*

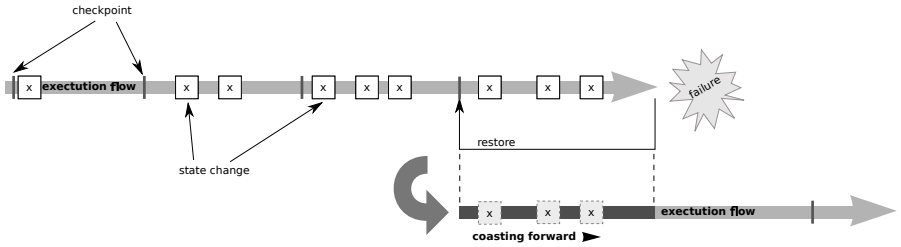


FIGURE 1.7: *Checkpoint rollback*

Needless to say that storage requirement may rapidly increases, accordingly to the application complexity. If, on the one hand, the CSS does not require any coasting forward, on the other hand, its disadvantages is a massive expenditure. To bound memory consumption, the fossil collection activity becomes fundamental.

Sparse State Saving Taking a new snapshot each time will overcharge the CPU with checkpointing scheduled operations. Sparse State Saving (SSS) is an optimization of the previous basic technique, which tries to reduce both storage and time overhead. Checkpoints are created either *periodically* or *adaptively* according to tunable parameters. In both of the aforesaid cases, program history is not "uniform" in time, on the contrary it results discrete. Therefore upon a rollback issuing, state is likely restored to a certain point in time and space. To realign the system, a coasting forward process is needed to rebuild the system state. Since each state change has not a related snapshot anymore, during the coating forward process, it must be ensured the computation will retrace the same previous executions trajectory. Non-determinisitc functions may divert the execution to another trajectory, bringing the system to a misleading final state. The correct behavior is firstly proposed in [**?**] as Piece-Wise-Deterministic (PWD).

The main difference between the Periodic State Saving (PSS) or Adaptative State Saving (ASS), is the algorithm which drives the checkpointing engine. As its name suggests, the former simply saves the program's state according to a checkpointing interval. A proper sizing of this parameter is fundamental to obtains the right tradeoff between the storage expenditure and the coasting forward duration. Adaptative technique, instead, relies on heuristic algorithm to dynamically adjust the aforesaid checkpointing interval to best approach optimum performance.

Incremental State Saving A further optimization of state saving is to strictly save punctual changes only, instead of the whole state [**?**]. The program's history is a form of "diff log" with respect to the last checkpoint. There are several implementation versions of the ISS, that herein we only list; for a further reading refer to [**?**].

Optimized ISS The original proposal

ISS with memory protection It adopts a memory security protection system relying on the OS features

Transparent ISS This version features a grater flexibility by allowing the overloading

Automatic ISS An implementation that relies on code instrumentation

1.3.2 Reverse code generation

The underneath idea of reverse execution is to lever the fact that many instructions are constructive and thereby state-conservative in nature (Table 1.2). Such a set of instructions can be easily reverted by computing theirs counterparts, e.g. `x = x + 1` is straightforwardly reversed

into `x = x - 1`. This method directly generates the reverse code of each constructive instruction. Generation process can be accomplished either(a) through a static pre-processing of the source code (b) or by dynamically emit the reverse code on-the-fly.

Static emitting The source code is statically parsed in order to create the program's control flow. Once the path-sensitive analysis has explored data and control dependences, the instructions are thus reversed, creating an inverse version of the program. An example of that approach is widely described in [**?**] by Akgul and Mooney. It is a SSA-based method which handles non-deterministic o destructive operations, i.e. branches or assignments, by embedding thrown bytes within reverse code. To distinguish which path to follow while backtracking the program, conditionals on those meta-data are used. To restore the value of a variable, they rely on combinations of the following three techniques described.

Redefine As the name suggests, a variable's value is restored by finding its nearest reaching definition of and recursively restoring the values it depends on.

Extract-form-use This technique embodies the proper code generation approach. It reverse executes an assignment command, whose function is invertible, by running the relative inverse function. Functions that involve more than one variable can be still inverted, provided that other variables are restore beforehand.

State saving Whenever the above mentioned techniques cannot be applied, the system falls back to the classic state saving (§1.3.1).

Memory overhead, in this case, is basically composed of the reverse code, which is statically predefined as the output of the initial parsing. Though, in multi-threaded environments and, in general, for highly non-deterministic programs, the overhead could grow unreasonably since a massive interleaving of threads and modules. It might be needed to consider a very high number of different possible paths from which unraveling, that would overburden preliminary parsing, if not thwart it. Further, since non-deterministic functions could not be always predicted —thus reversed— statically, this poses a second challenge to this strategy, which could be circumvented through the following dynamic approach.

On-the-fly reversion Instead of pre-processing the original program and statically building a reverse version of it, an alternative is to emit straightway at runtime the reverse code whenever the relevant operations are met, according to some criteria. In contrast to the static approach, it is well-suited for non-determinism since it exploits runtime information that the natural execution provides. Further, the memory would be more efficiently employed. The inverse code is generated dynamically on-the-fly, therefore the possibly untouched portions of code would not be generated at all, without introducing any additional overhead, neither. Nevertheless, since the middleware layer is in charge of producing the reverse code on-the-fly, it will inevitably slows down the forward execution in order to undertake its task. There is a threshold between adopting the static approach, which ensures a more slender process, and the dynamic generation which overcome non-deterministic issues at the expense of an extra time overhead. An example is given in [**?**, **?**].

The on-the-fly generation approach can be roughly accomplished both through a virtualized environment or by relying on code instrumentation, as we will do. The former approach requires that an on-line interpreter pareses the instructions to be executed and remap them into a simpler virtualized instruction set. However this solution is quite expansive, as one can easily imagine. The middleware layer heavily slows down the natural execution since it interposes the additional effort of the just-in-time translation. Hence the virtualzed environment must have much more computational power than the one being inspected. One advantage in adopting the virutalized solution is the employment of a reduced and simplified instruction-set which can be properly

devised to remap each destructive operation into a constructive one (see Table 1.2), simplifying the instruction inversion process.

To overcome the aforementioned performance hurdle, instead of running a virtual interpreter, our alternative is to instrument the code, in order to inject the assembly code chunks that act as "call hooks" towards the reverse engine. Once the reversing engine is invoked it can relies on the runtime information and data, which ease its task. This represents the novel approach towards which we are steered throughout this thesis, and the we try to embody with our module (Section 4.2). The reversion is accomplished in a two-passes process. First the code is instrumented, therefore the code injected invokes the real reversion engine which produce the inverse assembly instruction.

Specifically, our engine is in charge to produce at runtime the reverse code of instruction as it is met. Whenever those relevant instructions, according to some criterion, are met, the interpreter will produce the a block of code which realizes the inverse function. Likely the dynamic instrumentation approach, it is possible (§1.4) to fine-grain filter the instrumentation and thus the reversion process. Theoretically, the user can devise custom filters to produce reverse code only of a specific set of instructions.

1.3.3 The Hijacker's approach

Hijacker is a static binary instrumentation tool which relies on an proprietary intermediate representation of the input. The aim of this tool is to provide a wide-spectrum flexibility, featuring the support for several objects file formats and instruction-sets. The binary instrumentation at the machine level poses the following two main challenges; either a high machine dependence, and a user-side complexity due to the need to manually provide the code to instrument.

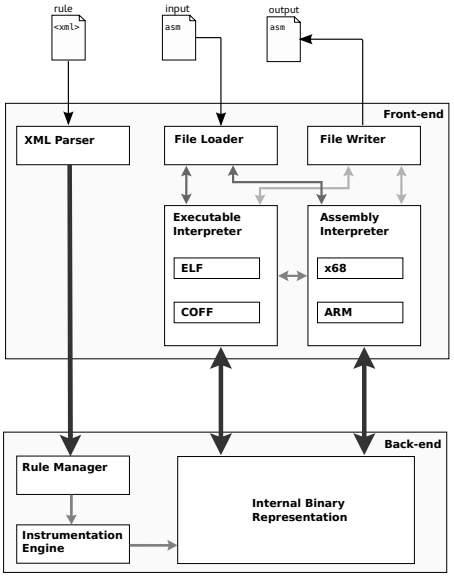


FIGURE 1.8: *Hijacker's high-level instrumentation schema*

In order to address these issues and to simplify source code handling, Hijacker maintains an intermediate representation of the original code (see Section 3.1 "Intermediate binary representation"). This internal binary representation allows to handle even complex instrumentation processes efficiently, allowing very extensible support to several executable file formats without the need to modify the inner engine. Further, the intermediate representation decouples instrumented instructions from the original ones, and abstracts code references between them. So

that it ensures the program's correctness throughout the instrumentation process and a complete compliance with respect to external compiler tools. Hijacker modifies this internal representation driven by a set of customizable rules provided through a xml file. A detailed description on the configuration rule file is given in Section 3.3.1. The rule-based structure of the file allows the user to customize the inner instrumentation engine which is thus highly modular and flexible according to the user needs. The user can either provides handcrafted code or chooses prepared code modules targeting purpose-specific aspects, such as memory inspection, I/O control, function call trace, etc. Such a way addresses the second challenge the instrumentation process poses, relieving the user from the mandatory and non-trivial task of devising assembly code.

Though Section 3.2 will provide a much more detailed description about the overall process, herein just an overview is given. Hijacker takes as input the relocation object file output by the compiler, afterwards the front-end will parse provided object files in order to disassemble its structure in basic blocks and mapping them to the corresponding representation. Once the object file is mapped to the relative program map, the rules provided instruct the engine on the specific instrumentation to perform. After the whole process is accomplished, the new object representation of the program will be recompiled and relinked with the rest of the software and the possibly employed extending modules.

We intentionally devise Hijacker to operate on the assembly level. Unlike higher abstraction levels, it does not provide any semantic information about the software logic. Hence, to reverse assembly instructions the instrumentation engine will eventually run into a challenge. By having no knowledge about the underneath semantic, its code perspective is a mere sequence of assembly instructions. The following example will explain better which is the real problem.

```
1  int main(int argc, char
     **argv) {
2    int n;
3
4    n = 8;
5    n++;
6    n /= 3;
7
8    return 0;
9  }
```

LISTING 1.4: *Source code example*

```
1   0:push   %rbp
2   1:mov    %rsp,%rbp
3   4:mov    %edi,-0x14(%rbp)
4   7:mov    %rsi,-0x20(%rbp)
5   b:movl   $0x8,-0x4(%rbp)
6  12:addl   $0x1,-0x4(%rbp)
7  16:mov    -0x4(%rbp),%ecx
8  19:mov    $0x55555556,%edx
9  1e:mov    %ecx,%eax
10 20:imul   %edx
11 22:mov    %ecx,%eax
12 24:sar    $0x1f,%eax
13 27:sub    %eax,%edx
14 29:mov    %edx,%eax
15 2b:mov    %eax,-0x4(%rbp)
16 2e:mov    $0x0,%eax
17 33:pop    %rbp
18 34:retq
```

LISTING 1.5: *Assembly code produced by GCC 4.9.2 idioms*

Consider the `C` code in Listing 1.4 example. The snippet realizes a simple function that makes some basic arithmetic operations on the variable `n`: an assignment, therefore an increment and finally an integer division, which is quite linear. Listing 1.5 which reveals the machine code behind the previous snippet. Although the simplicity of the operations involved, even a simple function is not straightway reversible just by linearly scan its machine code instruction by instruction. In this case, semantic would be compromised and actually guaranteed to be completely arbitrary. In fact, the parser engine must foresee logic operations embodied by groups of instructions, in order to correctly invert them. High-level logic is indeed necessary to properly treat instruction blocks, otherwise there is no way to predict how to pack instructions together, or to foresee which logic operation they would realize. A valuable research in this direction is faced by [**?**] in which the authors analyzed the possibility to reinterpret the assembly code back to the original high-level

logic. Though they provide a valuable tool able to correctly undertake the task, they likewise encountered several problems; which are actually the same issues Hijacker would have to deal with by handling the assembly code itself.

Each compiler, indeed, produces a different code ensemble that can be interpreted as the "fingerprint" of the compiler itself. The chief challenge is therefore that the same high-level statement, such as an integer addition, would be treated differently by different compilers, or even worst be different versions of the same compiler framework. This is quite upsetting, actually this means that once the source code has been translated into the assembly code by the compiler, along with the high-level semantic we lost likewise the possibility to backward reinterpret instruction blocks. But, why semantic is so important in the instrumentation stage? To answer this question is sufficient to look at the example 1.9 which well depicts the problem. The darker rectangle encapsulate the assembly instructions block that realizes the integer division by three. Machine instruction sets are much less concise than the high-level languages, and further no straightway division instruction is employed. A generic operation likely needs a block of several instructions, even though it is embodied by a single one statement in the programming language. Compilers are optimized to produce compact and efficient code, therefore the translation is not straightway plain. Every compiler generate a different sequence of instruction for the same logical operation. This is what in the literature is referred to as compiler's *idiom*. The knowledge of which compiler and version of it was used, may help in tracing back the operation correctly. However beside that this introduces a certain level of non-determinism, remains the problem of properly recognize logical operation from a bounce of assembly instruction. Another valuable contribute in this direction is given by the Tankut Akgul in his work [**?**]. He devised a powerful tool able to reverse instruction at the machine level. It employees a CFG (Control Flow Graph) in order to properly recreate the semantic beneath the assembly code.

We circumvented this problem by narrowing the set of reversible instructions to specific ones that would be easily "self-invertible". Indeed, our primarily objective in this thesis is to devise a reversible rollback support for parallel simulation environments. Therefore drove by this aim we concentrate the attention to exploits some caveats.

Non-deterministic parallel environments, such as multiprocessors or multi-threading systems and simulation platforms, would demand for an unfeasible amount of resources in order to keep

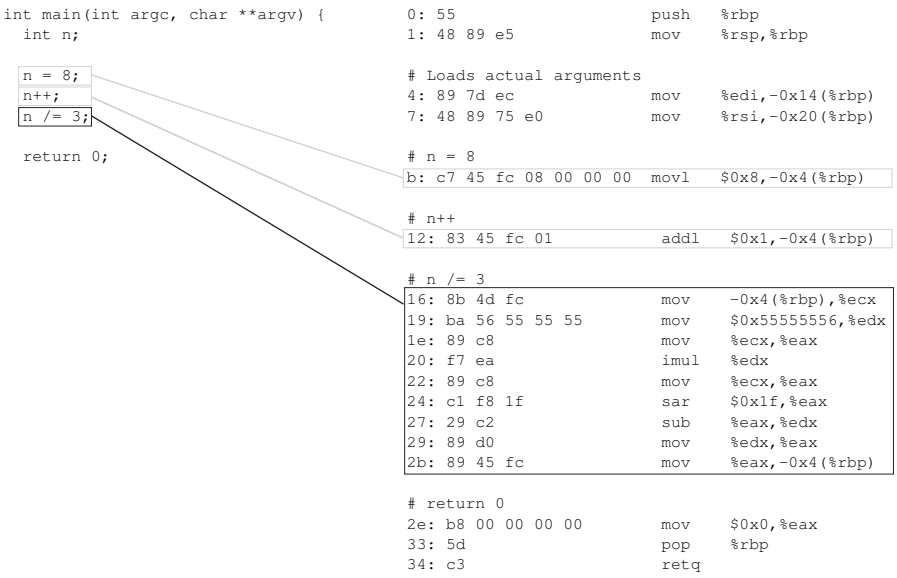


FIGURE 1.9: *Example of compiler's idiom*

track of the computation history. On the contrary in Hijacker, we have devised an hybrid approach consisting in dynamic generating a reverse code which is not the real translation of the native one, tough; rather it is a more concise form. A detailed digression is given in the further section Section 4.1 "The reversing code approach". For the reader's sake however, the underneath idea is to blend the benefits of the reverse execution together with the state saving approach to overcome destructive operation without entangling overall architecture. We further optimize the solution by narrowing as much as possible the number of instructions to be inverted. Our foremost objective is to support reversible execution in speculative simulation platforms. It does not need a fine-coarse precision for a punctual back-stepping, on the contrary simulation process require an efficient way to restore previous state values. Nevertheless our projects paves the way to a powerful and easily extendable solution also in debugging field.

1.4 Instrumentation

Some programmer considers a matter of pride to solve problems writing as few lines of code as possible, however this practice might result in even more cryptic code. Experience suggests to comment the code, to make it self-explainable. In the same way, as comments are used to assist code comprehension, instrumentation is the activity to monitor and profile program's performance or to diagnose possible errors by means of embedded instructions.

Instrumentation can be performed at source code or machine code levels, each one providing a different subset of information to rely on, as Table 1.3 shows. The former provides more semantic details which aids code comprehension and decouples it from the machine-specific instruction-set used. The structure of programming languages straightway provide the high-level perspective on the software logic, whereas assembly code level lacks of semantic information, which thwarts to infer which operation a block of instructions realize. However the assembly code has the advantage of ease architectural design of the instrumentation engine. Our strategy settles at assembly level to have a more direct control over extra instruction and data exchanged. Hitherto explained what instrumentation process is, but why it is useful? To answer to this second question a simple example will aid.

Let us consider a very simple car without any indication lights whose engine suddenly stops. There are several plausible causes, such as an engine failure due to high refrigerant's temperature, a spark plug breakdown, or rather gasoline has finished. However, trying to guess which is the right one is just as tossing up. Therefore, to solve the problem necessarily more information is needed either provided by the car's control system itself or by manually exploring under the hood. In the aforementioned example, since that car does not provide any information, what is left to do is to manually inspect the engine, deeper and deeper, until the failure is revealed, does not matter how long it will takes.

Actually this represents what a debugger allows to do. It aids the user to inspect instruction by instruction the whole code. However, coming back to the example, if our car would had at

	Source code	**Assembly code**
Level	High	Low
Detail	High	Low
Statement type	Complex	Simple
Structure	Structured	Unstructured
M. Dependency	Yes	No
Comprehension aids	Yes	No

TABLE 1.3: *Instrumentation's abstraction levels comparison*

least the gasoline indicator, probably one could easily understand that the tank is running empty and the problem can be solved in few minutes simply betaking ourselves to a gas station. The gasoline indicator represents why instrumentation in code is so useful, since even tough debugger allow to physically solve the failure, time and feasibility to locate it is completely up to what kind of information we rely on.

Definition 7 (Instrumentation)**.** *Instrumenting is the fine art of injecting purpose-specific code into a generic program (or environment) to achieve several objectives, such as providing relevant status information during program execution, tracing function or library calls or patching the software to arbitrarily change its behavior.*

What code instrumentation is indeed used for? The application fields to which the instrumentation process can be applied spans over a wide landscape from debugging to profiling and computer security, further it can be employed for several purposes:

- Execution flow tracing
- Data or function logging
- Software profiling and optimization
- Program patching
- Simulation and parallel application
- Error detection
- Automated debugging
- Testing and correctness checking
- Collecting metrics
- Binary translation
- Virtualization and emulation
- Deobfuscation
- Sandboxing
- Control flow and behavior analysis
- Malware analysis
- Vulnerability detection
- Reverse engineering

The aim of this digression was to underline instrumentation importance during software analysis. A very primitive form and example of instrumentation is the common practice of fulfilling the code with status output print statements. Although it is still useful and widely used, it cannot provide deep insights. Further it could be much inefficient since the code-flooding of debugging instructions may heavily affect performance. Extra code must be injected judiciously. Finding clever instrumentation algorithms is still an active research field [**?**, **?**, **?**, **?**, **?**]. Figure 1.10 depicts where the instrumentation process lies within the generic program's development workflow. Generally, instrumenting a target object involves to find proper points where injecting the own code, and taking control over the program. Therefore, the code has to be devised in order to properly save the program's execution context and restore it transparently afterwards it executes. Once the context has been restored, control can return to the program. Therefore Nevertheless,

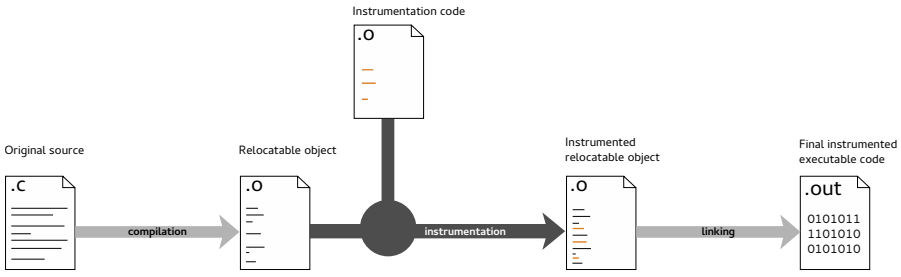


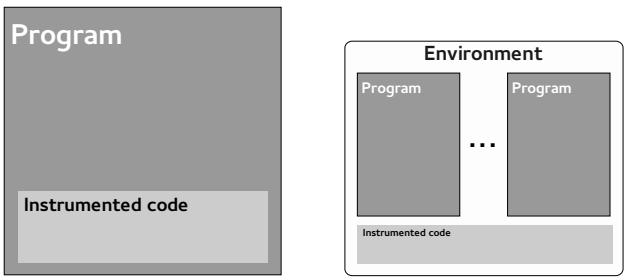
FIGURE 1.10: *General instrumentation's tool workflow*

the process may occur at several abstraction layer, therefore at a different developing stage: (*a*) at *hardware* level, (*b*) at *machine* level, (*c*) at *byte code* level (i.e. interpreted languages), and (*d*) at *source code* level. The former kind of instrumentation is the lower abstraction level. It leverages on real hardware elements or devices to interact with the system. It is used to commonly debug embedded systems, or low level devices that need a raw access to the electronic logic. On the contrary the machine level instrumentation rises up to the software level by acting straightway on the assembly code. Although it allows a tight control on the instrumentation by exploiting powerful machine-dependent tricks, it does not provide any suggestion on the high-level semantic. Bytecode instrumentation is quite similar to what is achievable with virtualization. Some high-level programming languages (e.g. Java) are not straightway compiled into the assembly code. The intermediate representation allows a virtual machine to interpret the code that are mapped at runtime to the machine-specific instruction set. This is chiefly done for portability reasons. Finally latter kind of instrumentation acts at the top abstraction edge on the source code. The instrumentation engine manipulates the high-level language (HLL) altering in place of the programmer. Source code instrumentation has been widely used in interpretative languages and has the valuable advantages to hold the program semantic; each operation can easily recognized and altered, further the same address space is shared, thus allowing to refer variable and symbols directly. However from an implementation perspective, it requires an additional parser able to create a non-ambiguous symbol map to work on. Furthermore, at this abstraction level fundamental information about relocation is not known until the early stages of compiling. Source and machine level binary instrumentation approaches are complementary. The former is platform independent and provides access to high-level information; on the other hand machine instrumentation is language-independent and can rely on low-level information which may be require for some tasks and furthermore does not need original code.

Software instrumentation is, thus, widely used to diagnose and optimize applications since it allows to collect a lot of inward information otherwise not available. It allows to identify bottlenecks, to estimate program's performance, such as branch mis-predictions, and to trace evolution path which is fundamental in symbolic software debug.

1.4.1 Ways to instrument the code

As previously mentioned, software instrumentation refers to code chunks which developers insert, directly or not, in the application in order to have a deeper insight, such as trace execution path, record values of inner variables, or profile the program by counting the number of specific function invocations, etc. Part of the instrumentation process is commonly handled throughout the design phase by developers, which insert code blocks within any area considered relevant, according to come criterion. Though, it is quite impossible to precisely foresee where debugging or profiling



(A) *Program instrumentation*

(B) *Environment instrumentation*

FIGURE 1.11: *General instrumentation's tool working schema*

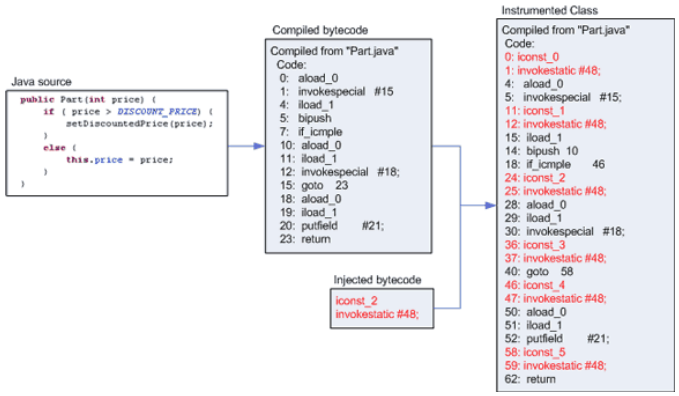


FIGURE 1.12: *Bytecode instrumentation example*

instructions may be useful, therefore the early software instrumentation has to be "handled with care".

Lots of tools address automatic instrumentation. There are two main ways to automatically instrument the code, *static* and *dynamic* instrumentation. The main difference relies on the time at which code manipulation is performed. Before to proceed, let have an overview about how it is performed from a high level perspective. Automatic instrumentation can be done at several levels of deepness, from higher programming language to machine-dependent instruction level. Figure 1.10 depicts how code injection is performed. The two approaches are complementary, each one providing its own advantages according to the specific purposes. Generally, static analysis involve correctness checking, optimization and performance-improving activities, whereas dynamic analysis is suitable for profiling and debuggers.

Static instrumentation

The static instrumentation approach inserts code at compile time by permanently altering the native program. Since, the insertion occurs at compile time, a recompilation step is needed each time debug instructions or custom redirecting functions are injected. This approach has better performance if compared to the dynamic one, since there is no extra computational effort required that will burden program's runtime. The only additional cost is the one needed to perform instrumented operations. Nevertheless, static instrumentation is less flexible. Since the instrumented code persists thoroughly the program execution, it does not support to dynamically switch on and off debugging code according to developer needs. Further, static instrumentation does not allow to instrument external modules like shared libraries, because would require to

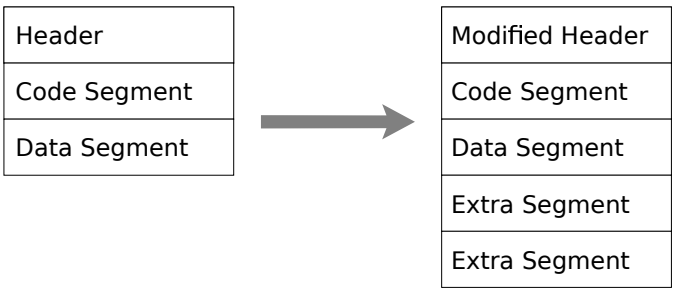


FIGURE 1.13: *Assembly instrumentation example*

recompile them too.

Dynamic instrumentation

Unlike the aforementioned paradigm, dynamic instrumentation injects the code dynamically at runtime according to a set of criteria. Dynamic binary instrumentation has the advantage to be reconfigurable at runtime, throughout whole program execution. It allows to enable or disable instrumenting features at runtime according to whether they are useful or not. Further, it usually does not require to prepare target program in any way, and allows to cover all client code. Since it executes at runtime, even dynamically generated code may be caught. Although dynamic instrumentation provides a more advanced and flexible insight tool, it introduces two main disadvantages, first of all an higher overhead due to the handling of the instrumented code itself, which affects runtime. Indeed, original code must be parsed, interpreted and matched against provided criteria in order to generate the instrumented code, which can be time-consuming. Second, it is not straightforward to handle executable code at runtime.

CHAPTER 2

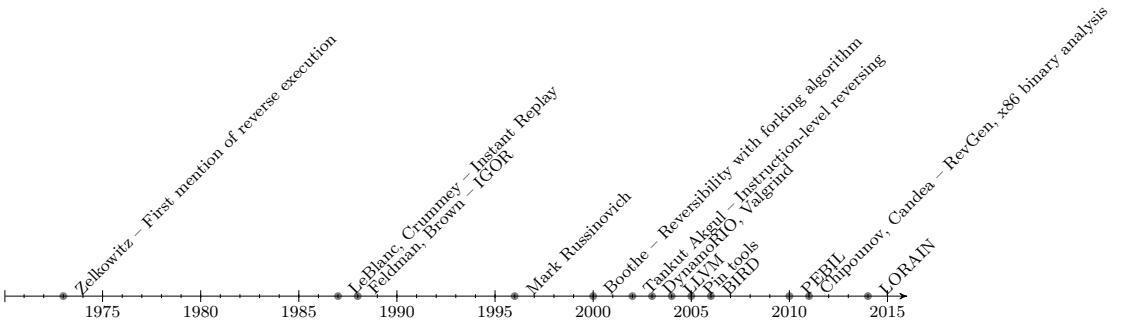
State of the Art

Contrariwise to what might be expected, program reversibility research is an ancient branch. The theory was born in the early 1970s, nevertheless it has experienced a new thrust rather recently, when the computational power and technology allowed it to grow up. Several works have been presented, from pure researches to the design of implementation tools targeting debugging or, generally, instrumentation purposes. What the present work pursues, is to overcome the most common problems that generally affect efficient reversible tool realization, specifically targeted to simulation processes. This section will give a survey on the related work's landscape, retracing logically the evolution beginning from one of the first reversibility notion appearance, and proceeding throughout the history by touching the more noteworthy works. For each one, we present the underneath architecture's overview, and furthermore which hurdles have been faced and how they have been solved. Beyond the overview on reversible debuggers, we focus the attention on the instrumentation tools, in order to give a detailed perception about where our actual work settles.

As further analyzed in the Section 1.4, the act of instrumenting the code alters the original one by injecting blocks of instructions to disclose useful information, which otherwise remain hidden. It spans from debugging scope, to performance evaluation up to software's security breaches analysis. Although several attempts which is possible to find around the matter, their diffusion as real working implementation is quite circumscribed. This is mainly due to an intrinsic architectural complexity and furthermore due to the burdensome computational and time overheads, if not prohibitive.

The aforesaid projects' objective will progressively drive throughout a novel instrumentation approach to finally touch performance evaluation of our framework as a mixed strategy for efficient rollback in speculative applications. Nevertheless, Hijacker is indeed a more general performance enhancer targeted to aid debugging and performance analysis tools by providing them advanced and modular features. Actually, for the sake of completeness, this chapter recalls what has been just presented by diving into a survey of proposed products towards a final section dedicated to the parallel simulation framework related work. Table 2.1 and Table 2.9 schematically report the highlights of each project.

Unlike debuggers endeavors —still more prototypes then real implementations—, advanced code instrumentation seems to be a more comfortable branch where several projects succeed in achieving thorough inspecting instruments, though onerous overheads. Valgrind (§2.2), for example, leaps among instrumentation tools as a powerful and valuable mean in order to inspect memory profiling secret chasms. Pin (§2.2) is another notable example of a complete binary instrumentation tools set developed by Intel ®, highly capable of inspecting inner aspects of beneath assembly. On the debugger side, instead, some interesting results were achieved by IGOR (§2.1) which works only on a specific machine instruction-set, or URBD (§2.1) which aims to create a universal middleware layer between the inquired program and whatsoever debugger the user chooses. A really fascinating outcome is presented by T. Akgul [**?**] where he deals straightway with instruction reversing problem, providing an efficient way to cope with.



2.1 Reversible and Post-mortem debuggers

Reversible (Post-mortem) debuggers exist since the 80's, thereby they are actually not a new matter of research. However, oldplatforms, characterized by poor computational capabilities, have prevented a real evolution of reversible debuggers which still nowadays introduce burdensome overheads. Further, at the time, there was not the real need to develop powerful instruments to debugrelatively simple programs. Nowadays instead, the growing software complexity brought to new landscapes. Parallel and distributed systems involve subtle interactions with several actors and software modules, which thwarts users to cleanly follow the logical execution flow. In the following we present some of the most relevant projects whose aims are to realize reversible debuggers, and in same cases even post-mortem, support. To give an idea we quote from [**?**]:

> [...] Faced with a stack overflow in a depth first search routine, or a related symptom, the programmer will examine the final state of the computation (the post mortem dump) and discover huge numbers of loops, though this will take a lot of work unless the debugger has a good way of picturing graphs. The programmer wants to ask:
> *"When did this tree first get an illegal cycle?"*
> The tricky parts of this question are: "when", "this tree", "first get", and "cycle". The first and third questions require being able to look back in time. The second requires identifying particular data structures, whose locations in memory will change with time but for which there must be some identifying property (location of the root of the tree perhaps). The fourth requires introducing a complicated test condition that is unlikely to be present in a standard debugger.

Instant Replay In 1987 LeBlanc and Crummey present an implementation of a repeatable cyclic debugger. Although this project does not represent a real reversible computation example, it is one of the first attempts to debug highly parallel applications. Those applications have quite complex interactions that render near impossible to directly focus what the problem is. Instant Replay save the events occurring during the natural execution only, without the related state changes. In this way, the authors guarantees a reduced space and time overhead. As a bug happens —or the program has to be rerun for some other reason—, Instant Replay debugger will reprocess saved events in the same exact order, preserving possible external inputs that might divert the execution trajectory. Instant Replay is not dependent to any form of intercommunication process, and avoids to any synchronization bottleneck by adopting a fully distributed algorithm. Further, Instant Replay provide a program-grain reply rather than process oriented one.

Project	Year	Basic technique	Highlights
IGOR	1988	State saving	Needs a modified compiler, library, loader and require to run on a patched kernel.
EPDB	2011	State saving	Employees a variant of the checkpoiting technique based on the forking mechanism provided by the operating system to create different timelines.
URDB	2011	State saving	Represents a universal reversible layer to enhance existing debuggers, based on a modified version of the program history.
RevGen	2011	Reversing instruction	A retargetable tool which interprets assembly code back to the LLVM's intermediate representation. It is not a debugger, but faces a quite similar challenge.
LORAIN	2014	Reversing instruction	Specifically developed for PDES platforms, it supports reversible execution by dynamically reverse constructive instructions.
LEONARDO	2000	Virtual machine	Supports program's reversibility and foremost its graphical visualization.

TABLE 2.1: *Instrumentation tools report*

The prototype implementation was developed on the BNN Butterfly, a tight coupled parallel processor comprised of 128 MC68000 units. Performance obtained are briefly presented in Table 2.2 below. The authors chose two applications for performance assessment, the Knight's tour and the Gaussian elimination,. In both of cases the overall slowdown is less than 5%.

Knight's tour A knight's tour is a path on a chess board for a knight that successively visits each square once and only once using legal chess moves.

Gaussian elimination Gaussian elimination (also known as row reduction) is a linear algebra solving algorithm for systems of linear equations. It is usually understood as a sequence of operations performed on the associated matrix of coefficients. Tested on a 400x400 matrix.

Application	Processors	Time (s)	Space (Kb)
Knight's tour	16	52	60
	64	43	48
Gaussian elimination	16	17	–
	64	20	–

TABLE 2.2: *Description of the two case studies for the Instant Replay debugger*

IGOR IGOR [**?**] is a prototype of reverse and post-mortem debugger developed on the Motorola 68000-based architecture[1] running DUNE [**?**] distributed system. The basic underneath idea which aims its authors, is basically similar to what Hijacker would provide. A system assumption is that program are written in C language, such that it can relies on a transparent calling mechanism and a rudimentary variable scoping.

IGOR's major goal is to restart program's execution by providing (a) either a post-mortem reviving program feature and a (b) reverse backstepping support during a legal execution by

[1]The architecture used supports the UNIX System V ABI semantic

adopting a checkpointing technique.. Former feature relies on the automatically generated memory dump from the operating system as the program crashes, it is targeted to parse the core file in order to restart the whole execution beginning at the first instruction statement beyond the faulty one. Needless to say that a core dump is not sufficient to guarantee a proper restart for some wayward failures. The second feature provides a backtracking support, it relies on a periodical checkpointing of the used pages within the program's address space. Process' execution is restarted by restoring the nearest checkpoint in memory, afterward an interpreter will forward re-execute code statements until some selection criterion is met.

The introduced overhead is quite expensive. Snapshots are taken periodically according to user's tuning parameters, but it could exceedingly grows depending on the software characteristics. IGOR does not handle the checkpoint much efficiently, and no particular optimization technique is really adopted, but narrowing the storage to the only dirty pages. Further it requires to use a modified version of compiler, library and loader over a patched operating system.

Table 2.3 shows the runtime performance for the *sort* testing program. The execution time of the modified compiler was 17% greater than the standard compiler for a typical source file, and 37% greater for the compilation and linking of a 47OO-lines program.

Checkpoint interval (seconds)	Execution time (seconds)	Overhead
None (baseline)	22.2	–
1.0	31.7	1.43x
0.3	37.3	1.68x
0.1	47.5	2.14x

TABLE 2.3: *IGOR runtime overhead for the* `sort` *program*

	Real time (seconds)	User CPU (seconds)	System CPU (seconds)
Regular execution	31.9	17.6	1.2
Restart time	54.5	5.7	6.9
Portion in user program	10.1	3.9	0.3
Overhead	44.4	1.8	6.6

TABLE 2.4: *IGOR restart time overhead for the* `sort` *program*

EPDB EPDB (Extended Python DeBugger) [**?**] is an extension of the yet existing reverse debugger for Python programs. It supports post-mortem inspection of core dumps and further allows to step backwards along the program execution path. This represents a new feature if compared to the aforementioned tool, even though also EPDB provides a reverse execution feature by adopting a very similar snapshot&replay technique. Snapshots are achieved by forking the running process. Each fork generates a new timeline, the former is paused in order to be retrieved in future, while latter branch is instantly resumed transparently. Through timeline approach, EPDB guarantees to efficiently handle dynamic functions of instructions, such as user-input ones, by creating a new branch whenever the user wants to alter the input provided, otherwise the original one is rerun. Performance of EPDB is shown by Table 2.6 by taking a sample set of five programs as described in Table 2.5.

URDB Universal Reversible Debugger (URDB) [**?**] is a framework layer which enhances debugging tools by providing a uniform middleware layer for reversibility. Currently, it provides support for the following four debuggers: (*a*) GDB, (*b*) Python debugger, (*c*) MATLAB, and

Name	Description
fankuch	Takes a generic permutation and computes all the possible n other permutation, where n is the first element of the input permutation.
n-body	Models the physical problem of solving gravitational interactions between celestial bodies.
gcd	calculates the greatest common divisor for two very large integers using Euclidean algorithm.
call_snap	Models the insane scenery where the program repeatedly call a non-deterministic function.
create_array	The program will create a very large integer list array.

TABLE 2.5: *EPDB performance table*

Program	Runtime (s)	Debugging runtime (s)	# Snapshots	Base Memory usage (MB)	Debugging Memory usage (MB)
fankuch	0.034	37.7	13	3	44
nbody	0.051	366.4	65	3	106
gcd	0.115	49.5	10	3	37
call_snap	0.029	8.6	501	3	1010
create_array	0.93	0.98	3	766	785

TABLE 2.6: *EPDB performance table*

(*d*) Perl. URDB interposes itself between the user and the debugger intercepting checkpoint or reverse commands. In those cases the framework acquires the control and performs relative actions. Again, also URDB relies on checkpointing techniques, whose engine is realized through an enhanced version of the DMTCP (Distributed MultiThreaded CheckPointing) supporting `ptrace` utility. URDB provides a "reverse wrapper" for each of the basic debugging command, (*a*) `step` (*b*) `next` (*c*) `continue` (*d*) `finish`, in the form of `reverse-`*`cmd`*, where *cmd* is one of the above forward commands. Like the aforementioned tools which rely on checkpoiting approach, again reversibility is achieved by restoring the last checkpoint to which follows a forward instructions re-execution[2]. Each URDB's checkpoint collects a debugging history. Therefore, analogously to the aforementioned tools who rely on checkpoiting approach, every time a reverse commands is issued, (i.e. `reverse-`*`cmd`*), the framework clones the last checkpoint's history and produces a new history to run in order to place the user at the proper point in time.

In [**?**], performance has been evaluated by taking 4 debuggers as benchmark, classical GDB v7.2, Matlab, Perl, Python. Table 2.7 suggests that URDB is fast enough for the interactive use in reversible debugging.

Command	gdb v7.2	Matlab	Perl	Python
checkpoint	1.86	2.02	0.17	0.18
restart	1.20	1.65	0.20	0.17

TABLE 2.7: *URDB time overhead in seconds for checkpoint/restart*

LLVM LLVM (Low Level Virtual Machine) [**?**] is not properly a debugger, instead it represents a compiler collection, similarly to what GCC is. The underneath LLVM's idea is to provide a

[2]The forward re-execution process is also known as *forward coasting*

universal compiler relying on an intermediate representation (IR) in order to completely decouple architecture dependencies. LLVM, hence, is a complete collection of tools from compiler through optimizer up to the analyzer, which aims to make lifelong program analysis for arbitrary software in a transparent way. LLVM exploits (*a*) an internal code representation which decouples it from architecture's and language's peculiarity and (*b*) a compiler design which properly leverages this representation. The internal LLVM's representation (IR) is a RISC-like instruction set relying on the Single Sate Assignment (SSA) technique in order to eliminate register ambiguity. Because of its characteristics, LLVM could be considered as a complement to high-level virtual machines, not an alternative to them.

RevGen Again, also RevGen [**?**] is not properly a debugger, though it is designed in order to translate x86 code into LLVM Intermediate Representation (IR). The underlying idea is to support reversibility for external IR-based debugging tools. We report this work as it has to cope with very similar challenges we encountered in assembly-level reversing code generation: (*a*) extracting binary code's semantic and (*b*) inferring type information. Instruction level, as discussed in later chapters on this thesis, does not provide any indication on how to interpret a generic assembly block. In RevGen, the authors proposed an interesting technique steered to face *retargetable* compilation. That is, to interpret assembly code into an another representation; in this case, the LLVM's IR.

RevGen firstly translate the input code into a LLVM translation block (TB), representing a sequence of micro-operations. This is subsequently splitted into basic blocks (BB) that allow to finally build the control flow graph (CFG) relative to the original binary code. The CFG allow to infer the high-level semantic. RevGen, as LLVM, relies on the Single Statement Assignment (SSA) to remove symbol reference ambiguity. Each variable's name is translated into a unique one, therefore a reversible approach can be adopted.

LORAIN LORAIN[3] specifically designed for PDES (Parallel Discrete Event Simulation) [**?**][4]. It relies on the LLVM's intermediate representation and supports a rollback without the state-saving technique by emitting the reverse code. LORAIN leverages the fact that many operations which alter the state are *constructive*, and therefore reversible. In order to properly handle destructive instructions, overwritten related variable's value is saved. However, since not all the operations are constructive, for those that dispose useful information during the execution, LORAIN resorts to classical state saving. The intuition is to undo altering instructions instead of rollbacking to a prior checkpoint.

LORAIN makes the assumption that only instructions that directly affect the memory can produce a state change. In those cases, it reverse the CFG, build via the LLVM architecture, along with the basic block path taken during the forward execution. This ensures to correctly reverse the `if` statements, for example.

The results demonstrate that the generated models are able to execute at a valuable rate that near the on obtained with the hand-written model. Specifically, the authors considered three version of both PHOLD and Airport models:

- (O0) Unoptimized
- (O3) Optimized
- (HW) Hand-written

The following Table 2.8 depicts the results in terms of events per seconds.

[3]LORAIN stands for *Low Overhead Runtime Assisted Instruction Negation*, so it is not the Marty's mother as in Back to the Future.

[4]The authors used as PDES framework the Rensselaer's Optimistic Simulation System (ROSS) [**?**].

Model	O0	O3	HW
2 core PHOLD	946,164.80	971,834.45	974,658.30
32 core PHOLD	6,070,813.30	6,155,326.35	6,178,446.40
2 core Airport	1,921,355.20	2,039,775.40	2,062,018.85
32 core Airport	6,713,835.70	6,854,997.75	7,044,712.05

TABLE 2.8: *LORAIN's performance*

LEONARDO Leonardo [**?**] is a visual representation tool designed to enhance debuggers capabilities by providing a real time visualization of the actual program's state. Along with visualization, reversibility is supported by employing a virtual hardware system which fully support instructions reversibility. LEONARDO realized reversibility similarly to what LORAIN does, without relying on checkpoints.

2.2 Software instrumentation approaches

Instrumentation process can be *static* or *dynamic* according to whether code injection takes place in compilation or linkage stage, respectively. Generally, the goal is to deviate the natural execution flow through control statements or interrupts traps towards some instrumented function which, in turn, performs whatsoever inspecting purposes; afterwards control yields back to the native code. However this approach will inevitably incur in overheads, that are sensible to the abstraction level at which the instrumentation is preformed.

To recap the Section 1.4 "Instrumentation", static instrumentation introduces a restrained overhead since the burdensome part of the work is performed before program get running, hence without hurting execution performance. Nevertheless, it provides less features compared to the dynamic instrumentation which, rather, may exploits runtime environment's information. However, the dynamic approach will clearly introduce an heavy overhead, since the tool interposes itself as a sort of middleware. It parses all the instructions and generally translates them into a more convenient intermediate representation.

In the following, Table 2.9 reports a brief highlighting schema of each project considered in this section, emphasizing the instrumentation approach adopted.

PEBIL PEBIL [**?**], or rather PMaC's[5] Efficient Binary Instrumentation toolkit fo Linux, as its name suggests, is static binary instrumentation tool which insert a branch instruction to debug code at the entry point of each functions. PEBIL provides a set of APIs to insert a lightweight custom assembly code, rather than relying only on basic instrumentation utilities.

PEBIL was designed for Linux platforms to handle ELF executable format only, either x86 and x86_64. It first analyzes the whole source code in order to interpret text code and separates it form data. This code's discovery algorithm can be either *control-driven* or a *naïve* linear disassembler, which exploits the symbol table to identify each function's entry point. Though, the engine is not fully deterministic, in fact PEBIL's disassembly algorithm reach a coverage of 99.0% on the CPU2000 Integer Benchmarks. Once the code is properly interpreted, the PEBIL's instrumentation engine creates an extra text segments containing debugging code along with a possibly new data segment.

Likely other tools, PEBIL inserts a branching call instruction towards the instrumentation function at the entry point of the native one. Therefore upon the function call, the control passes to the instrumentation code which typically performs some debugging and/or performance

[5]PMaC stands for Performance Modeling and Characterization.

Project	Year	Instrumentation type	Highlights
PEBIL	2010	Static	Developed for ELF format only in the Unix-like environment, it employs a double (control-drive or naïve) algorithm to disassemble instructions for entry point function finding.
BIRD	2006	Static	Built for Window/x86 standard binaries, exploits code trap by injecting a 5 bytes jump instructions to hook instrumented code; if 5 bytes are not sufficient an interrupt is used.
DIOTA	2002	Dynamic	Uses a double code approach. It generates a clone copy by dynamically emitting reverse code without altering the native one; in reverse mode the new copy is used.
DynamoRIO	2003	Dynamic	It adopts instructions caching techniques in order to guarantee a fast translation of frequently executed basic blocks.
Pin	2005	Dynamic	Intel® framework providing a wide range of inspecting tools. It is based on Just-In-Time injection strategy.
Valgrind	2003	Dynamic	Multipurpose tools, according to the "skin" dressed, basically providing memory inspecting analysis.

TABLE 2.9: *Instrumentation tools report*

operation: (*a*) saves the context register, (*b*) executes debug code and (*c*) finally restores the context register along with the control given back to the native function.

Figure 2.1 illustrates a compared chart of the PEBIL's performance achievable throughout a set of benchmark programs. As can be seen PEBIL's overhead is bounded, if compared with the other tools.

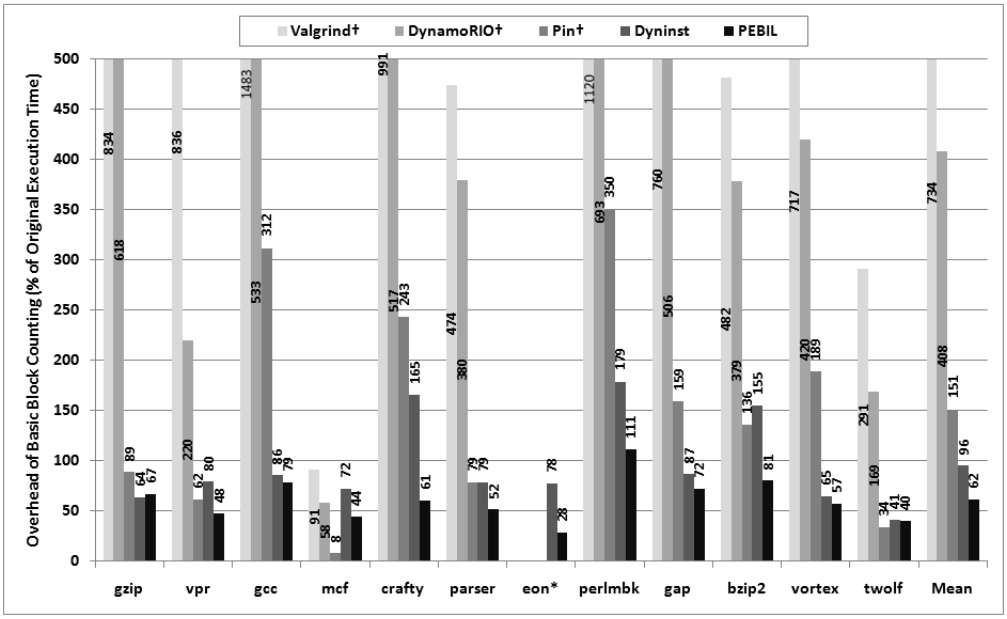


FIGURE 2.1: *PEBIL's performance compared chart*

BIRD Built for Window/x86 standard binaries, BIRD [**?**] relies on a hybrid instrumentation strategy, both static and dynamic and does not require debugging information. In order to divert the execution flow, BIRD uses a similar approach to the aforementioned PEBIL by adding branch instruction towards a debug function. Those uncontitional jump instructions take 5 byte at the instrumentation point to be inserted, however space is not always enough, therefore, unlikely PEBIL, BIRD employs an interrupt trap. Before the program is executed, the *static* disassemble engine tries to cover as much code as possible of the text section as known areas; once the program is running, the unknown remainder of code is processed by the *dynamic* disassembler. BIRD adopts a double-staged disassemble process to interpret instructions from data which build a confidence score, beyond a certain threshold a binary unit is definitively considered as code rather than data.

As in [**?**], Table 2.10 presents BIRD's effectiveness expressed by the following two parameters:

accuracy Measures the error in correctly interpreting the instruction for block of bytes recognized as code.

coverage Represents the fraction of bytes in the binary file that are not recognized neither as code nor as data.

Further the Table 2.11 presents an overview of the overhead that BIRD introduces for a set of common programs.

Application	Coverage (%)	Accuracy (%)
lame	96.7	100
ncftp	84.4	100
putty	96.1	100
analog	88.7	100
xpdf	86.1	100
make	95.5	100
speakfreely	70.0	100
tightVNC	74.9	100

TABLE 2.10: *BIRD's effectiveness table*

DIOTA Authors of DIOTA (Dynamic Instrumentation, Optimization and Transformation of Applications) [**?**] specifies that its name derives form Latin and refers to an old Romans double-handled drinking vase, which well describes the underneath idea the tool relies on. DIOTA instruments code dynamically by emitting code on-the-fly but leaving unaltered the native program. It progressively generates a clone, thereby during the program's execution, binary code is picked from the clone whereas the data is read from the original source. In such a way, DIOTA ensures the instrumentation correctness even for hard-to-instrument programs, containing data in code and vice-versa. Since DIOTA maintains code separate from data it is not possible they hurt to each other or the overall semantic. DIOTA's approach guarantees to properly treat either code embedded data and polymorphic program at the same time.

DIOTA's authors state that their framework is relatively fast. It introduces a slowdown ranges from 20x up to 400x. When using memory or code instrumentation, the overhead naturally increases, especially with the former since in that case every instruction that accesses memory is accompanied by a call to the installed callbacks.

DynamoRIO DynamoRIO [**?**] is an evolution of the prior Dynamo [**?**] project developed for Linux/Windows x86 platforms. The goal is to provide a very wide application field, spanning from debugging through optimization up to emulation. DynamoRIO is a framework for implementing dynamic analysis and optimizations. It parses and manipulates program's instruction at runtime, transparently, adopting an instructions caching techniques in order to guarantee a fast translation of the frequently executed basic blocks. It further exports an API constituted by a wide set of functions and data to manipulate binary instructions. The basic idea of DynamoRIO is to support not only instrumentation traps but also to allow a flexible and dynamical manipulation of the code.

Application	Runtime (s)	BIRD's runtime (s)	Static instr. overhead	Dynamic instr. overhead	Total overhead
comp	0.068	0.086	15.1	0.1	15.2
compact	3.671	3.907	6.4	0.0	6.4
find	2.657	2.825	6.2	0.0	6.2
lame	0.425	0.479	12.0	0.0	12.0
sort	0.093	0.111	17.5	0.4	17.9
ncftpget	0.379	0.389	3.4	0.0	3.4

TABLE 2.11: *BIRD's performance table*

Pin Pin [**?**] is an architecture independent instrumentation systems which provide a set of APIs in order to profile, optimize and inspect programs. It relies on a very similar model as ATOM which is built only for the Alpha architecture. The Pin distribution provides several Pintools including profilers, cache simulators, trace analyzers, memory bug checkers and allows access to architecture-specific information. It adopts the Just In Time compilation strategy to inject and/or optimize code and heavily relies on several optimization techniques in order to guarantee valuable efficiency. Unlikely the aforementioned DynamoRIO or Valgrind, Pin aims to be fully automated relieving the user to manually hand write specific functions.

Valgrind Valgrind [**?**] is a program supervisor framework which relies on dynamic instrumentation, supporting the ELF file format and running on most of the x86 Linux environments. It is structured in a twofold architecture,(a) a main *core* and (b) a series of pluggable modules, namely *skins*. Those "skins" represent specific-purpose sub-tools that can be selective plugged to specialize the instrumentation process. Out of the box, Valgrind comes with several default skins:

Memcheck A purify-style memory checker for C and C++. Traces all the memory accesses.

Addrcheck A more lightweight memory checker which only checks whether the referenced address is within the addressable space.

Cachegrind A cache profiler which traces instructions and data cache accesses and misses.

Helgrind A data-race detector that uses the Erase algorithm [**?**].

Nulgrind A "null" skin that performs no instrumentation at all.

Valgrind's *core*, on the contrary, contains the main support for the program instrumentation. It is constituted by a set of wrapper libraries, a scheduler and a just-in-time (JIT) compiler. It translates original x86 instructions into an intermediate two-bytes opcode representation (*UCode*) to simplify program handling. The UCode representation is expressed in term of virtual registers. After an initialization setup, Valgrind will not run any part of the client program on the real CPU, rather the execution is moved on a simulated environment. The just-in-time compiler translates one basic block at a time, where a *basic block* is a set of instructions which ends upon a control-transfer instruction, such as a `jump`, `call` or `return`. Each basic block which needs to be optimized is disassembled in the UCode instruction-set, instrumented, mapped to the real registers and finally translated back to the native x86 code. Signals are handled by diverting traps to the Valgrind's own handler which adds signals to a pending queue. The delivery of those signals is performed periodically by Valgrind, except for the non-resumable POSIX ones.

Valgrind is a customizable framework, allowing hand writing of custom skins according to the user needs. Nevertheless, for this reason it could be prone to larger overheads, depending on the developer cleverness. Table 2.12 depicts the slowdown introduced in various program for each of the previous default skin. As one can observe, Valgrind is a quite expensive tool, though it highly depends on the actual skin used.

Program	Runtime (s)	Nulgrind (ratio)	Memcheck (ratio)	Addrcheck (ratio)	Cachegrind (ratio)
bzip2	10.7	2.4	13.6	9.1	31.0
crafty	3.5	7.2	44.6	26.5	107.4
gap	0.9	5.4	28.7	14.4	46.6
gcc	1.5	8.5	36.2	23.6	73.2
gzip	1.8	4.4	20.8	14.5	50.3
mcf	0.3	2.1	11.6	5.9	18.5
parser	3.3	3.7	17.4	12.5	34.8
twolf	0.2	5.2	29.2	18.5	53.2
vortex	6.5	7.5	47.9	32.7	88.4
mean	–	**5.2**	**27.8**	**17.5**	**55.9**

TABLE 2.12: *Valgrind's slowdown ratio with regards to the runtime*

CHAPTER 3

Reference instrumentation tool

As already mentioned, Hijacker is a framework oriented at providing the necessary supports to generic code alteration, specifically in High Performance Computing applications. It addresses this issue by implementing a *static* instrumentation at the assembly level, which augments the executable code to generate at runtime the reversing instructions according to user needs. Although *dynamic* instrumentation is more flexible with respect to the static one (Section 1.4 "Instrumentation"), it is much more time-expensive, and therefore clashes with the objective to allow a feasible implementation of reversible execution in High Performance Computing (HPC). Nevertheless also the static approach has its disadvantages, such as the impossibility to directly instrument third-party libraries; although, it can be overtaken by some tricks.

Within the software developing process, Hijacker lays between the compiling and linking stages. It hence operates on the intermediate phase onto the *relocatable object* files. Such a choice allows to decouple the architectural details of the target machine from the Hijacker's constructive ones, therefore ensuring the solution to be portable and flexible. Section 3.1 "Intermediate binary representation" investigates implementation details of the binary representation, by which we tried to build a uniform abstraction layer for severals different formats and operations. The choice to work on relocatable representation of executable files was mainly driven by the following two motivations: (*a*) the availability of low-level information which ease the construction of the internal representation by giving much more control on instructions, data and cross-references; and secondly (*b*) because it much simplifies the emit phase, relieving it from the need to explicitly resolve relocation details. Linkable[1] format, by definition, provides no semantic but conveys instruction-grained details —e.g. relocation placeholders— through which is straightforward to build the IBR. It is basically constituted by basic blocks, the same provided by linkable files. On the other hand though, working on relocatable representation of executable files increases the user-side difficulties. Relocation resolution issues are left to the classical linker. In such a way we do not only lighten the Hijacker's architecture but we foremost ensure modules compliance, with regards to the environment, and software correctness, either syntactical and semantic.

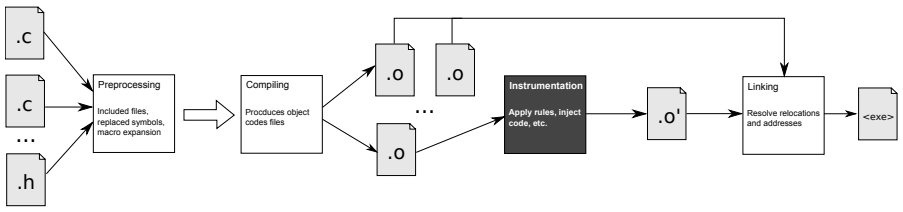


FIGURE 3.1: *Hijacker's stage*

Hijacker takes as input a relocatable object file which is parsed in order to create a program

[1]Terms *linkable* and *relocatable* are used interchangeably.

map maintained by an internal binary representation (IBR). By means of that representation, Hijacker keeps the relevant references needed to navigate the whole code throughout its instructions and to dynamically alter it as required (§3.1). The instrumentation is realized by a rule-driven engine (§3.3) whose directives are provided by the user through a xml file (§3.3.1). Those rules drive the engine in tweaking the input program acting straightway on the internal representation; for example, by injecting additional custom instructions or code blocks, or rather by diverting the original control flow through user's wrappers. Section 1.4.1 "Static instrumentation" previously discussed about the typical drawbacks of static paradigm, such as the impossibility of dynamically enable instrumented features or to instrument third-party libraries and modules. Compared to the dynamic approach though, much of the computational overhead introduced will burden only the pre-processing stage rather than hurting the runtime program's performance. Aside from for debugging purposes, it is a fundamental requirement for real-time parallel applications.

Finally, Hijacker will rebuild the output file from the altered the IBR, which holds the instructions' references needed to remap it back onto the object format. Such a file, is again a relocatable object entirely compliant with any compiler framework; thus allowing to easily re-link it together with the remainder of the software. In this way, modules can be separately instrumented and linked according to developers needs.

By adopting an internal representation, binary code is decoupled from the intrinsic limitations yielded by the Object File Format (*OFF*). Because of the offset-based fixed structure, working right on relocatable files is hazardous and it might likely results in an incorrect output. On the contrary, Hijacker rebuilds the object file from scratch, ensuring consistency (§3.2.2) is kept in a safer way. Relocatable files have, in fact, a structure purely based on displacements in order to realize cross-references among units of storage (see Chapter 7 "Appendix C"). Therefore, altering them would require a burdensome effort to keep offsets consistent within sections and data. Consequently the overall complexity of such an architecture would be quite uncomfortable and complex to handle.

Hijacker is composed by of two modules: a *front-end* which directly interacts with the user, interprets and emits objects files, and a *back-end* that is basically constituted by the rule manager and the instrumentation engine.

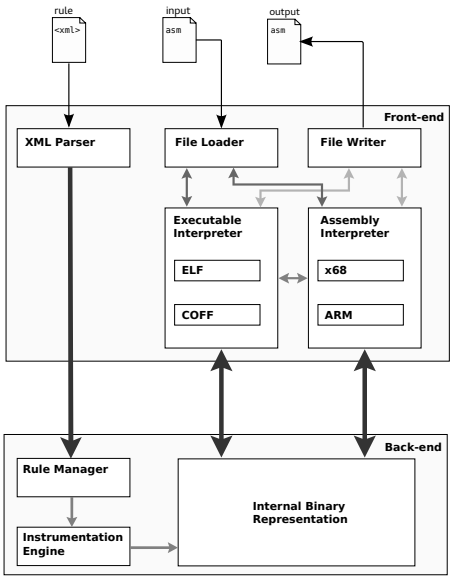


FIGURE 3.2: *Hijacker's architecture. Front-end provides file parser and emitter supporting vary relocatable object files; rule manager and instrumentation engine is in the back-end*

As depicted in Figure 3.2, the front-end is in charge of reading the input file. Therefore it translates the relative program's structure into the IBR by interpreting the underneath software's logic and the interactions among its parts. File parser resolves each symbol and each relocation reference among instructions, or between instructions and data, and embeds the relative information. Basically, Hijacker represents the executable code as a chain of instructions, which in turn, are maintained by a logical structure holding cross-reference data; namely, the descriptor (Figure 3.6).

Conversely, the back-end basically comprises the rule manager and its counterpart, the instrumentation engine. Each rule is interpreted by the manager and therefore applied to the aforementioned binary representation. Given the linear, and likewise dynamic, structure of the program map, the instrumentation can be straightway performed. Whenever a rule is applied, the references among instructions, functions and data are preserved since they are realized as logical pointers. Instructions can be simply added to (or removed from) the existing chain leaving unaltered all the unrelated links. Nevertheless, an update process is still required in order to realign some of the node's fields within the representation, such as the nominal address of each instruction. So far, back-end is the module who introduces the major overhead due to progressive renew of each address reference field; however this one may be considered negligible since it is a pre-processing submerse cost.

The instrumentation process at the assembly level, although allows a grater compatibility and portability, it likewise requires more effort to provide all the necessary modules. By operating at the machine level, it involves a strong dependency with the relative instruction-set, further it also shifts a considerable complexity degree on the user side. Indeed, to instrument at such an abstraction level, the user itself would be asked to provide the code to inject, such that it would also be compliant with the target architecture. A task anything but harmless, since lot of underneath details the user ought be aware of, such as (*a*) the instruction-set itself (*b*) the architectural design (*c*) the specific ABI (Application Binary Interface) implemented. Nevertheless, Hijacker still allows users to build custom instrumentation codes via specific rule tag (refer to Section 3.3.1 "XML configuration rule file"), provided that code is straightway handwritten in the assembly language. However, since what we just hinted before, Hijacker comes with a set of precooked modules to address the most common instrumentation issues, such as the reversing-monitor module. This is specifically designed to support the reverse execution, which implementation deserves a dedicated dissertation (§4.2). Nevertheless to give the reader an overview, our module is conceived to reverse instructions that affects memory. In this way, we aim to optimize additional requirements. The module is based on an hybrid instrumentation strategy that blend together static and dynamic aspects. During a first static instrumentation stage our framework adds specific assembly blocks which are functional to dynamically invoke the reverse code generator. Further, we adopted a mixed reversibility approach which interweaves both state-saving and reversible intuitions.

3.1 Intermediate binary representation

Hijacker builds a program map of the executable file relying on a internal *ad-hoc* intermediate representation. Adopting an internal representation, it decouples technical and architectural details of the underlying machine from the semantic high-level code aspects. In this way we are able to easily construct the map by treating cross-references within the code, simply as memory pointers. Thereby there is no need to care about of relative offsets since they will be correctly rebuilt in the final emit stage. ELF format, as the other relocatable files actually, has in fact an offset-based structure which is quite awkward to cope with.

By parsing the input file[2], the front-end maps it into such an intermediate representation,

[2]At the time of this writing Hijacker framework supports only the ELF format, though other supports are

namely the *internal binary representation* (IBR). Each instruction is disassembled by the proper interpreter such that its meta-data can be retrieved and stored into a logical high-level structure, which are doubly linked together such that the whole program can be easily navigable, back and forth. Instructions are split into separate blocks —each representing a function— only in a further step of the file parsing, when symbols and references will be available. Once the references among instructions, and between instructions and data are resolved, they are therefore linked to the relative *symbol* descriptor, that keeps the pointer to the target *instruction* or *data* structure.

In such a way, code interconnections are decoupled form the native object's section they belonged to, which much simplifies the instrumentation process and the manipulation of the code to be altered. Through the internal representation, relocation entries are resolved as memory pointers maintained by the descriptor. Therefore the appliance of the instrumentation rules will result in a straight process of inserting and removing —or otherwise modification— nodes within a high-level linked list of descriptors. Logical memory pointers ensure that target descriptors will be always consistent throughout the whole process. Figure 3.3 depicts the logical process of inserting an instruction (the removal one is specular).

By altering the IBR structure, for example by inserting a new node, it has likewise necessary to update the `new_addr` field with the new logical address the instruction actually have taking into account the new just inserted. The `new_addr` member can be seen as the shadow of the future position that instruction will have in the output object file. Further, the update of each real instruction's address requires the instrumentation engine to check the following issues: (*a*) that jump instructions have enough space to embed the new offset as immediate value and (*b*) to update any static references to data sections. Since the address information is intentionally not computed as relative displacement from the virtual address, it needs to be recalculated each time a new insertion or modification is undertaken.

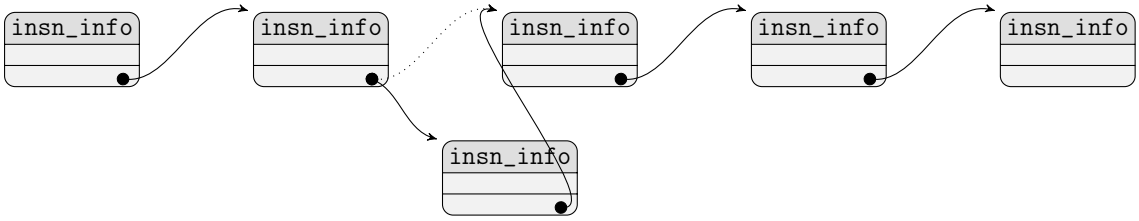


FIGURE 3.3: *Example of adding an instruction to Hijacker's binary representation*

Our aforesaid binary representation is not the LLVM one, as discussed in the chapter Chapter 2 "State of the Art", but it is an *ad-hoc* structure. Likely to most of similar tools, this representation is proprietary; though this could be a hurdle in merging features belonging to different tools. Figure 3.7 depicts very well the structure of the Hijacker's binary representation.

Funding elements of the Hijacker's internal representation are the following

- Instructions
- Symbols
- Functions
- Raw data

Hijacker represents the object input code, basically, as a chain of instructions. Each instruction in the chain is wrapped by an high-level descriptor which holds the relevant information

under development.

needed to navigate throughout the whole code. Basically, the instruction descriptor keeps trace of the disassembly code itself and a pointer to its symbol reference, allowing to easily rebuild relocation displacements in the future emitting step. That reference pointer can be either another instruction, such in case of a jump instructions, or a relocation symbol. Actually the IBR does not employ real relocation entries, rather it realizes them by means of pointers among the high-level infrastructure. This would give a better flexibility and efficiency in instrumentation. In fact, relocation is simply handled by cloning the target symbol descriptor to which relocation applies and embedding it into the relative instruction descriptor (§3.2). In such a way, it is possible to arbitrarily alter the code without the need to recompute each time all the displacements. Aside from the computational waste in which this method would turn otherwise, it furthermore could hurt syntactic correctness of the emitted object.

Symbols Symbols are organized in a simple linked list. A logical descriptor is composed of a sequence of meta-information, from its name to the position it occupies in the object file. Since a symbol may refer several kind of entities, position field provides a fast way to associate it back to a relocation entry as needed. This allows to dynamically alter code and it still guarantees to recompute relative offsets between the instruction and the symbol to which relocation applies.

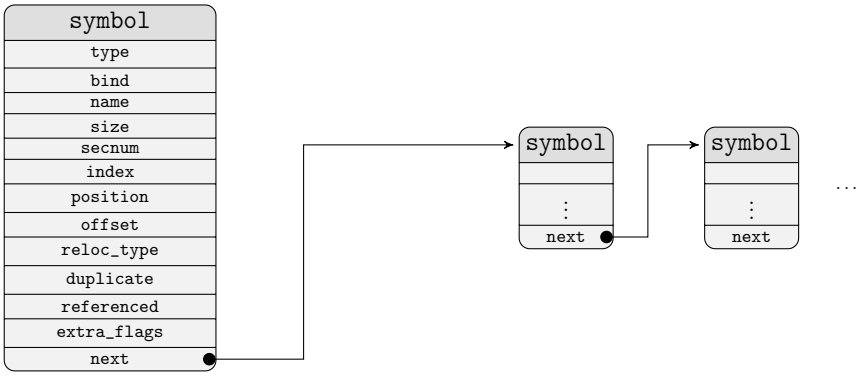


FIGURE 3.4: *Symbol descriptor*

Table 3.1 provides a brief description on each field of the symbol descriptor.

Functions Function descriptors basically hold the pointer to the entry point from which it begins and the reference to the relative symbol it belongs. Functions are basically used in the back-end by the rule manager in order to easily trace the instrumentation process. To simplify the user side, rule specifications are functions-based, as Section 3.3 depicts.

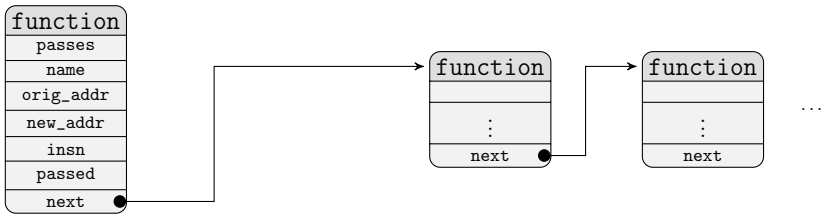


FIGURE 3.5: *Function descriptor*

Table 3.2 provides a brief description on each field of the symbol descriptor.

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

TABLE 3.1: *The instruction descriptor fields table*

Field	Type	Description
`passes`	`int`	Number of instrumentation passages that the rule manager has to undertake for that function.
`name`	`char *`	The function's name.
`orig_addr`	`unsigned long long`	The original code logical address of where the function begins in the object file.
`new_addr`	`unsigned long long`	The logical address the function would have in the instrumented object output file.
`insn`	`insn_info *`	Pointer to the first function's instruction descriptor of the doubly linked list that represents the function's body.
`passed`	`Function **`	Pointer towards the reference of the instrumentation engine function descriptor which realizes the rule.
`symbol`	`symbol *`	Pointer to the relative function's symbol descriptor.
`next`	`function *`	Pointer to the next function descriptor in the linked list.

TABLE 3.2: *Function's descriptor table*

Instructions Instructions get stored into a logical descriptor which holds the relevant information regarding the one just parsed. That structure actually is folded into two abstraction layers in order to hide the instruction-set dependency (*a*) a high-level descriptor, referred to as *instruction descriptor* and (*b*) a machine-dependent descriptor, namely the *disassembly descriptor*. The field `i` of the instruction descriptor embeds the machine-dependent structure, which maintains the original disassembly byte-code along with several attributes describing the instruction's specific implementation; such as the invocation flags, SIB bytes, immediate values, etc. The instruction descriptor, instead, holds generic information needed to perform logical operations on the binary representation. At the same time, the disassembly descriptor is useful in order to perform complex manipulation during the instrumentation process. Further a `reference` field is reserved to hold the memory pointer of a multiplexed type of target descriptor. In fact it realizes connections between instructions or between instructions and data; that are due to code branches, function calls or data movements, respectively. Specifically, the `reference` field may keep a pointer towards another instruction, or rather to a function's symbol. Raw data values are treated by wrapping them into a stub section's symbol descriptor, which will be relocated to the proper data section by the emitter.

Field	Type	Description
`flags`	`unsigned long`	Collects the actual *family flag* the instruction belongs to (§3.2).
`orig_addr`	`unsigned long long`	Holds the original address the instruction has in the native code.
`new_addr`	`unsigned long long`	Holds the actual address the instruction has throughout the instrumentation process.
`size`	`unsigned int`	Maintains the size of the instruction itself.
`opcode_size`	`unsigned int`	Holds the size in byte of the whole instruction but the possible immediate value or embedded offset.
`i`	`union::<info_insn_xyz>`	Embeds the machine dependent low-level descriptor.
`reference`	`void *`	Holds the memory pointer towards the target to which refers its relative relocation (if any).
`prev`	`struct instruction`	Maintains the pointer to the previous instruction descriptor.
`next`	`struct instruction`	Maintains the pointer to the successive instruction descriptor.

TABLE 3.3: *The instruction descriptor fields table*

Each instruction descriptor holds a pointer either to another instruction descriptor or to a symbol one, according to which kind of relocation applies, namely the `reference` field. This allows to maintains connections within the virtual representation itself. However, it does not to be forgot that the Hijacker's internal representation have to interface to both a virtual descriptor and the raw offset-based structure, as the one of the object file. Hence, the fields `orig_addr` and `new_addr` keep, respectively, the logical address the instruction naively has in the object file, and the one it will have in the instrumented one. Those fields are used as identifiers in both the parsing and emitting stages. However, one should not confuse the address fields with the reference one, though they seems to be in contrast with each other. In fact, via the `reference` memory pointer —specifically used to denote a virtual descriptor— the instrumentation engine can navigate the executable's binary representation, to finally provide to the emitter the logical address held by the `new_addr` field. In order to ease the output file generation, the emitter

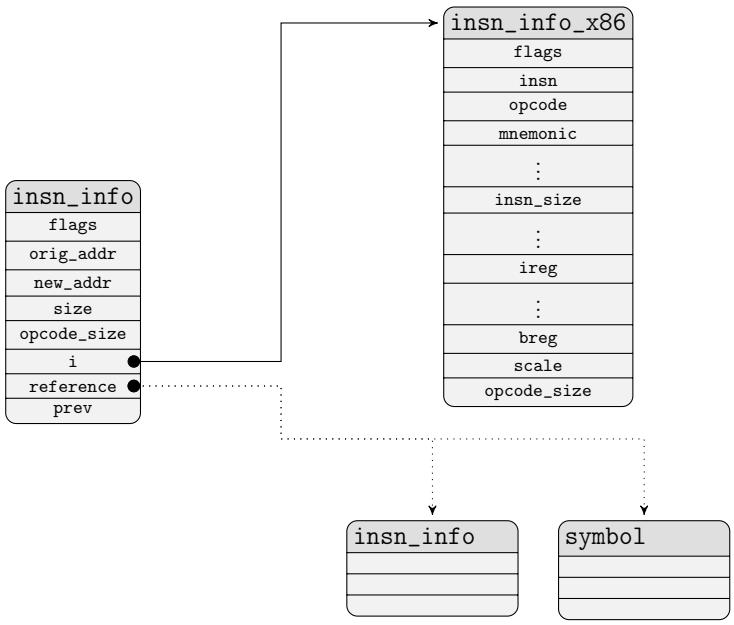


FIGURE 3.6: *The instruction descriptor*

uses the relative field to embeds offsets into the assembly code, instead of recomputing them. Otherwise, addresses have to be computed on the fly each time. In that case, the addresses may be solely retrieved from the relative distance between the instructions in the chain. Therefore, the emitter should count how many "hops" separate one instruction from the other and should take into account the size of the inner ones. A quite expensive task.

3.2 Front-End

Front-end module is constituted by the *file loader*, which parses the input relocatable file, and by the *file writer*, which emits the instrumented version. Further, the parsing engine is designed to support several object file formats (OFF), from ELF to PE and COFF, ecc. Therefore the front-end is also equipped by an executable interpreter and an assembly one. Those two subsystems are triggered by the file reader whenever a machine-dependent stage has to be undertaken, allowing to parse combination of object formats and instruction-sets. Each of the aforesaid modules is bidirectional since the they serve either in the read stage and during the emit phase.

As mentioned above, the front-end takes as input the relocatable object files. This is an intentional choice intended to give more flexibility to hijacker which lays directly between the compiling and linking stage. In this way the instrumentation engine can rely on a wider collection of low-level information. Section 1.4 overviews several instrumentation's abstraction level. The assembly level, even though it provides less semantic information about the logical program's structure, indeed allows a deeper analysis of the interrelation among instructions and their references.

The file loader module is in charge of reading the configuration file, containing user-defined instrumentation rules, and instructs the rule manager on the back-end. Therefore it reads the input file to retrieve its native format. Currently Hijacker fully supports only ELF relocatable format, though are yet under development other formats, such as PE and COFF. According to the format detected, the loader triggers the relative executable interpreter which starts to decompose the object file's structure by analyzing each section. The output of this process is a

binary representation, also referred to as *program map* (§3.1). Virtually it is composed by (*a*) a code section, realized as a chain of instruction descriptors, (*b*) a data section which keeps trace of program's referenced data and finally (*c*) a sequence of raw sections. The latter kind represents those sections which do not have to be instrumented; therefore they are straightforward rebuilt in the following emitting step. Along with the format information, input file parser will also retrieve the proper machine *instruction-set* employed. According to it, when a code section is found, the parser will invoke the relative disassembly engine. Currently, Hijacker provides a x86/x86_64 disassembler.

Once triggered, the assembly interpreter linearly scans the code section's content to translate it as a sequence of instructions. Bytes within the section are progressively parsed by the disassembler engine in order to interpret them as a sequence of instructions. Whenever the disassembler correctly interprets a new instruction, it builds the relative descriptor. As previously observed, it holds original instruction bytes (in raw bytes) along with a set of attributes describing the instruction itself in a more convenient way. Figure 3.6 provides a snapshot of the machine-dependent descriptor. Further each instruction will be marked with a local *family flag* which delineates its behavior (Table 3.4). Any of the previous flag can be OR'ed in whatever

Name	Description
I_MEMRD	The instruction reads from memory.
I_MEMWR	The instruction writes to memory.
I_CTRL	The instruction performs some check on data, such as `test`.
I_JUMP	Family of `jump` instructions which alter the execution flow.
I_CALL	Instructions who call a different function
I_RET	Instructions who return from a callee.
I_CONDITIONAL	The instruction performs only if certain condition are met (e.g. conditional jumps).
I_STRING	Instructions who operates on strings.
I_ALU	The instruction perform logic/arithmetic operations.
I_FPU	The instruction perform floating point operations.
I_MMX	Instructions who use MMX registers.
I_XMM	Instructions who use XMM registers.
I_SSE	Instructions belonging to the SSE instruction-set.
I_SSE2	Instructions belonging to the SSE2 instruction-set
I_PUSHPOP	`push` or `pop` instructions.
I_STACK	The instruction operates on memory within the stack.
I_JUMPIND	Instructions who realize indirect branches.

TABLE 3.4: *Family instruction flags*

combination (e.g. I_MEMRD|I_MEMWR is an instruction that either reads and writes on memory). Those families are used by the rule manager to specify where to focus and how to perform the instrumentation process.

During the assembly scan, the interpreter checks each instruction cross-reference, temporarily stored for a later pass when is ensured that symbols are parsed. In the further passage the executable interpreter will effectively resolve and therefore links each instruction descriptor with the relative reference. By this way, Hijacker generates a structure depicted in Figure 3.7. The instruction descriptor maintains a pointer to a multipurpose reference field, which is linked by the disassembler either to another instruction descriptor or rather to a symbol descriptor. Once the relocatable file is fully read, the control passes to the back-end, and therefore to the rule manager. This latter will invoke the instrumentation engine module relative to each rule encountered, which in turn will alter the internal representation. A deeper insight on the instrumentation stage will

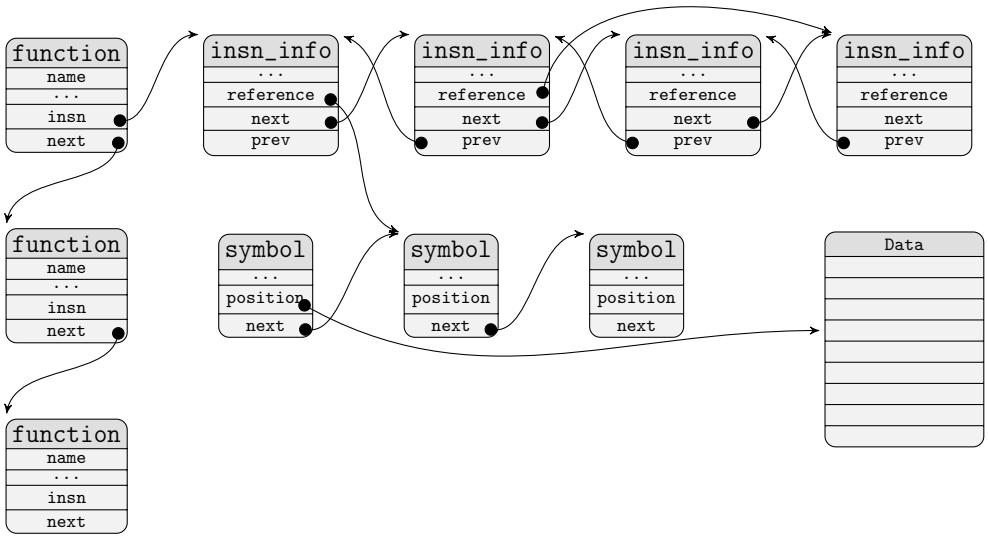


FIGURE 3.7: *Hijacker's internal representation*

be given in the following section about the back-end (§3.3). The control returns to the front-end as soon as the rule manager has properly instrumented the program map; thus a new relocatable object file will be generated from the altered program map (according to the input's format).

The output relocatable file is fully compliant to any compiler frameworks, (e.g. GCC), allowing to relink the newly instrumented module together with the remainder of the software. In this way, instrumentation can be modularized providing more efficiency and flexibility in its appliance. Each software's module can be instrumented in a different way according with developers needs and linked as required.

Shared libraries or other third-party external module support is only provided by allowing developers to create custom rules which injects hand-crafted assembly code. Those rules can be used to divert library invocation through a user-defined wrapper, even though it is not possible to instrument the library itself.

3.2.1 Object file

Object files are concise representations of applications' attributes, providing an efficient access to that information needed throughout the compilation process.

We intentionally chose to work on relocatable object files in order to maximize the extensibility of our tool and to further simplify the code handling process. By adopting this approach, the instrumentation engine can rely on machine-specific information otherwise not provided by any higher abstraction level. Although the relocatable representation lacks program's semantic references, on the contrary it provides a tight control over the assembly instructions. For the specific purpose of instrumentation at machine level, having a precise view about underneath details of each instruction is fundamental, since it allows to act directly on them. Unlikely the source level instrumentation (Section 1.4.1 "Ways to instrument the code"), assembly manipulation focuses the intervention producing an optimized instrumentation.

Executable which holds code and data suitable for linking

Relocatable which holds code page-aligned executable segments

Loadable binary code to be dynamically loaded and embedded in other programs

For the sake of completion, this section traces down a brief overview about the object files (actually, for a deeper insight in the ELF format refer to Chapter 7 "Appendix C"). Conforming with the information they provides, object files may be *relocatable*, treated by linker to produce executable ones, *loadable* as in case of external libraries that have to be embedded into a program, or *executable*. According to the specific purpose, the object file contains a structured set of information needed to perform the required tasks. Generally, those objects are structured in sections, that are segments of various size containing precise kind of data belonging to the following list. Relocatable objects are generated by the compiler as the output of the first compiling stage, (see Figure 3.1) they will be the future input for the linker. This one, in turn, extracts from the relocatable the relevant information to compute relocation's displacement for each entry and finally builds an executable file. In general, an object file thoroughly comprises the program, that is structured in different cross-referenced sections.

There are several kinds of format, each of one storing slightly different information either in content or in structure. Nevertheless, they basically have to keep track of a predefined set of meta-data.

Header information Overall specifications about the file size, its initial address (in case of executable files), and furthermore all the references to reach each of the other sections composing the object.

Object code Herein is stored the binary instructions of the assembled source code

Relocation Cross-reference section which bind symbols together with target code address where linker has to intervene adding dynamic references.

Symbol table A table listing local and global symbols, either functions and variables, belonging to the current module dynamic libraries and external reference to be imported.

BSS data Uninitialized static variable storage unit

Debugging information Source file line number, symbols and other meta-data employed by the debugger to visualize to the user program's execution flow.

Object files may be *relocatable*, which are treated by the linker, *loadable* as in case of the external libraries that need to be dynamically embedded into a ready-to-run program, and finally *executable*. According to the purpose, an object file contains structured information. Generally, object files are structured in sections or segments containing precise kind of data, aligned to the previous description list (Figure 3.8). Relocatable (or linkable) files basically hold sections containing data and a not-yet-complete binary code, along with a list of relocation sections. This file is suitable to be linked with other relocatable object files to produce executable files, shared object files, or other intermediate relocatable objects. The relocatable format is the output of the compilation stage, therefore it makes a very extensive use of symbols and relocation entries as placeholders to fix up addresses and references known only at linking time. Shared libraries and external symbols belonging to other modules, for example, belong to this category. Through a relocation entry the linker resolves instruction references and offsets into real addresses or displacements with respect to a unique entry point address, assigned to the executable by the linker itself. The output of the linking stage is thus an executable file, which holds needed information to load the program's map in memory and run it. Executable files, unlike relocatable ones, do not have to store much detailed information, since they are self-contained and ready to be executed. Therefore, they are no more than a page-aligned set of code segments which can be loaded and straightway run.

Our project currently supports only the ELF format [**?**, **?**] which is the one most used in Unix environment. Other file format supports are currently under development, such as COFF and PE.

In order to give a more valuable view, in the following a brief insight about the ELF file format is presented. Our choice lands on the ELF format because of its broad diffusion within Unix-based environments —which is actually the one used to develop this project. An ELF (Extensible Linkable Format) file[3] is quite flexible and extensible structure not bounded by any processor or architecture details. It is realized by different sections that are linked together via offsets, as Figure 3.8 shows. Generally, an ELF file consists of a variable number of sections,

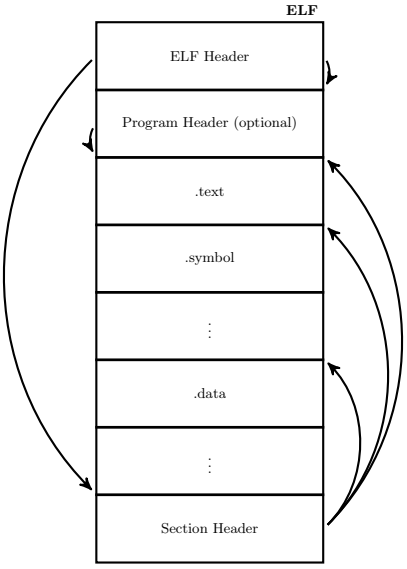


FIGURE 3.8: *An overview on the ELF structure*

accordingly to the specific characteristics of the program it represents. As its name suggests, the ELF file may represents either a linkable or a relocatable file, respectively. According to the type, the overall structure changes a bit. Both types are characterized be an *ELF header* block at the very beginning of the file. The header contains several attribute that specify the format itself and the architecture the program has compiled for. It represents a "road-amp" to describing the file organization. The relocatable type is further composed by *sections*, containing the intermediate data to allow the linker in solving relocation references; whereas the former type is only composed by *segments* containing the binary code, aligned to the page size for a more efficient memory loading Further the ELF header holds the offsets to one optional *program header* and to the *section header* table[4]. Former section is mandatory only for the executable type, since it maintains the necessary information to create the process image, to load it in memory and to launch it. Section header table, in turn, contains information describing the file's sections. The table maintains an entry for each section registered. A single entry holds the offset to the relative section's content along with several attributes describing itself, such as the section size, the content's type, the section name, etc.

Sections are referred to as *storage units* and hold plain binary data. The interpretation of that bytes depends solely on the section's attributes kept in the relative table's entry. have meta-data associated to them, further maintained within the header of the ELF file. The previous list cited the most common ones.

In the following, a brief description of the common sections.

[3]ELF format is firstly published in the System V Release 4 ABI, and in the 1999 it becomes the official standard for the Unix-based systems

[4]The section header table always closes the ELF structure

Header Which contains generic specification about the ELF file itself, such as the architecture for which the file is compiled to, the version number, etc. Further it holds fundamental information about how many sections or segments are registered and where the relative table is located.

Program header table Provides information about how to create the process image when requested its execution, such as the entry point address.

Sections Each section contains different kind of data according to its own type:

- `.text`
- `.data`
- `.rodata`
- `.bss`
- `.strtab`
- `.symtab`
- `.rel/rela`

Section header table Provides information to properly handle each section, how to reach it and what it contains, according to its type.

3.2.2 Parser of the relocatable object

So far, we have provided an overview of Hijacker, a sketch on its architecture and how it works. This chapter, on the contrary, focuses on the object parser and its emitter counterpart, giving a richer description about they are implemented.

Both of them retrieve the input file's format in order to properly interpret the information they provide. Herein, as previously mentioned, the focus is on the ELF format. Once the proper executable interpreter is invoked, it linearly parses sections one by one. To our purposes, basically only `text`, `data` and `relocation` sections are considered, in order to build the internal binary representation (§3.1). Other sections, generally do no contain relevant information for the instrumentation stage, and do not have to be instrumented themselves, therefore Hijacker will mark them as `raw` sections. A raw section is straightway rebuilt in the emit step by leaving it completely unaltered. In order to properly build the IBR, more passes are needed. The ELF structure is very compact in design, thereby information stored are deeply coalesced; they must be progressively rebuilt:

1. Linear scan to disclose sections
2. Resolve symbols and binding them to the corresponding entity
3. Resolve relocation entries linking the virtual descriptors

In the first step the input file is thoroughly scan, the parser stores each section's content into a partial representation that will be refined in future passages. The output is a temporary list of virtual section descriptor. Hijacker marks each entry with a internal flag wrapper in order to identify the type of the relative section, to create an abstraction layer. Table 3.5 describes the bind between ELF section's flags and the Hijacker ones. According to the section type different actions have to be undertaken.

Text section ELF's text section are marked by the `SHT_PROGBITS|SHT_EXECINSTR` value held by the relative section header (see Chapter 7 "Appendix C"). Whenever such flags are encountered the parser interprets machine dependent information in order to discover which instruction-set has been used. Then, control pass to the disassembler engine which linearly scans and decomposes all the raw bytes into instruction descriptors. Those descriptors are doubly linked together in a instruction chain. So far, instructions have been correctly interpreted and stored

Flag	Description	ELF Flag	ELF Attributes
SECTION_CODE	Section containing executable code that may be instrumented	SHT_PROGBITS	SHF_EXEC
SECTION_DATA	Section contains plain data it can map either read-only section and read-write ones	SHT_PROGBITS	SHF_ALLOC and/or SHF_WRITE
SECTION_NAMES	Section that contains strings of literals, such as section's names or symbol's names	SHT_STRTAB	*none*
SECTION_RELOC	Relocation sections, either REL or RELA type are coalesced into a single type	SHT_REL SHT_RELA	*none*
SECTION_RAW	Section irrelevant to build the binary representation and/or has not to be instrumented, rebuilt as is	SHT_NOBITS SHT_DYNSYM SHT_HASH SHT_DYNAMIC	*none*

TABLE 3.5: *Binding between the ELF's section flags and Hijacker wrappers*

in the representation. No functions are detected, and no reference at all are known. Only in a subsequent pass this information can be properly retrieved.

Once the whole file is read, Hijacker leans to all necessary information to identify and split up each function into a sub-chain of instructions. Simultaneously Hijacker's parser checks for jump instructions, which are linked to the target one descriptor. In this way, it is possible to correctly rebuild the reference network even when those instructions are altered or displaced as consequence of the rule applying phase.

Relocation Relocation entries are the only mean through which it is possible to properly rebuild the reference network among instructions. Therefore, great attention must be paid in their handling. Beside short jumps, every other reference unknown at compile-time has to be relocated at link-time. Since the alternate of the version and updates, relocation sections can either be represented by `rel` or `rela` entries. Basically, they differs form each other in the presence of an explicit structure's member (for a detailed description refer to Chapter 7 "Appendix C"). Whenever the parser meets a relocation section, it generates a temporary list of all its entries. The list is functional to bind instructions and symbols together in future passes. However, we still don't have enough information to solve relocations[5].

Once the parser has read the entire object, partial binary representation holds enough information to allow relocation to be correctly interpreted. Function `resolve_relocation` is in charge to check, entry by entry, at which address they refer to among instruction descriptors piking up the correct one. Instruction descriptor has a member (check Figure 3.6) to hold a

[5]We refer to the *solving relocation* action in a broad sense meaning to bind each target instruction descriptor to the relative symbols descriptor, within the internal binary representation.

Flag	Description	ELF Flag
SYMBOL_FUCNTION	The symbol is associated with a function defined within the user software	STT_FUNC
SYMBOL_VARIABLE	The symbol is associated to an data object (e.g. variable, array, structures, etc.)	STT_OBJECT
SYMBOL_UNDEF	The symbol's type is not otherwise specified; it can be an external library function or an `extern` symbol	STT_NOTYPE
SYMBOL_SECTION	The symbol refers to a section; those symbols are used in relocation to displace within a specific section	STT_SECTION
SYMBOL_FILE	The symbol conveys the name associated with the current object file	STT_FILE

TABLE 3.6: *Symbol flags binding*

generic reference pointer to the target descriptor, either instruction or symbol or, rather, raw data. Thereby, current instruction will be linked to the symbol descriptor to which the relocation applies. This allows to arbitrarily alter that symbol ensuring correctness in the rebuild stage. In fact, references are computed from the new addresses, maintaining the correct relative displacement.

Data section To our scope, data section type generally merges together `.data`, `.rodata`, and `.bss` section types which fall into the *raw* category, along with others special sections, such as `.dynsym`, `.hash` and `.dynamic`. No particular operation has to be performed on those sections, since they are not directly involved in the instrumentation process. Nevertheless, data sections are not ignored, but indirectly referenced into the internal representation and throughout the parsing process. As depicted in the Figure 3.7, data sections contains global variables or other (un)initialized data that code references through the relocation entries. Therefore raw sections need no special treatment but to be "embedded" into the binary representation as memory pointers to the proper target descriptor.

Once all the sections are found, the second step is to resolve symbols. As for section's type, also symbols a flags are remapped into a more convenient abstraction level. Table 3.6 shows the possible bindings of each ELF's symbol flag into the internal one. Generally a symbol may be either a *function* or a plain *variable* —in fact, the other types are not strictly relevant for the instrumentation aims. In case a new function symbol is met, a new function descriptor is created and linked to the relative symbol descriptor so that it can internally be referenced. Instructions chain is linearly scan to find function's extents in order to cut the chain down. Simultaneously whenever a jump instruction is identified it is linked to its target. At the end of this pass, each function has its own sub-chain of instruction representing the body. Otherwise, if the symbol represents a variable, the parser resolves it by displacing within the target section containing its data; therefore that value is linked to the symbol descriptor in the internal representation. The Hijacker's binary representation profoundly hinges on the notion of symbol. Basically, symbols are used as conveyors of the relocation notion, to substantiate references among entities in the representation. Indeed, the final step is to resolve relocation entries saved into a temporary list of descriptors. Once references are thoroughly translated, this list is disposed. This choice is aimed to create an abstraction layer to decouple object's entities from the notion of relocation. Because of hard-coded offset, relocation entries are unpractical to handle in for our instrumentation purposes indeed, therefore we collapsed it into the symbol descriptor itself. Nevertheless, one symbol descriptor ought be duplicated whenever a relocation entry to it is found, otherwise different relocation to the same symbol would overwritten by the last. Further, since instruction

and symbol descriptors are linked together, each instruction descriptor whatsoever modification of that symbol, would affect many other instruction as well; hence the reference to the symbol descriptor must be unique.

3.2.3 Hijacker's Emitter

So far Hijacker's IBR has correctly been structured as a chain of cross-referenced instructions. Control is thus given to the instrumentation engine driven by the rule manager. Each rule has been previously parsed and scheduled for the future instrumentation step. Now, the representation is ripe to be tweaked, as the following section faces. Once the instrumentation is done, control returns to the executable interpreter which in turn will invoke to the code emitter. The file emitter will output a new relocatable file perfectly compliant with the external linker so that it can properly build the remainder of the software modules. In order to ensure correctness, the emitter rebuilds the ELF structure from scratch. A set of canonical sections are recreated starting by default. Nevertheless, currently Hijacker does not support multi-text section objects, meaning that this type of file, will be reconstructed by straightway coalescing all the native text sections in a single one. Emitter starts by creating the stubs of the canonical sections which are subsequently filled up by successive passes. Data sections are straightway written in the new container, as well as string tables. Whereas symbols and text sections require a more refined treatment.

Symbol list maintained by the program's map is not the real one that will be written on the file, it is rather a virtual representation. Symbols, in fact, have been cloned in order to settle references towards instructions. Since more references can be made to a single symbol, each descriptor is cloned, so that the instrumentation be slender. The structure maintains the same semantic and allows to correctly identify the relocation reference in the emit step. Though only one copy for each symbol will be written in the output ELF file.

Instructions are scanned linearly during the emit phase, for each one the emitter will invoke the specific machine code emitter. Raw bytes packed into the instruction descriptor, will be written to the corresponding target section, in the meanwhile the emitter checks jump, call or other type of references in order to build the relative relocation entry to write out along with the instruction itself.

3.3 Back-end

Back-end is constituted by a rule manager which drives the instrumentation engine. Once the internal binary representation is successfully created by the front-end, the rule manager will instruct the inner engine to properly instrument the code according with the rules provided by the user in the configuration file. The configuration file is simply an *xml* file through which the user can directly drive the instrumentation process by choosing functions or instructions to be added, diverted or muted. The user can straightway specify instrumentation modules to pin or handwritten assembly code blocks to be injected. In this way the user is able to create arbitrarily custom snippets to achieve more efficiently his/her objective. A more detailed description of xml rules format and usage is provided in the further section (§3.3.1).

The rule manager will scan the configuration file and will apply each rule in the order it would be met. A set of internal functions allow to add or substitute low level instructions within the IBR. In the following a brief description.

`insert_instruction_at()` It creates room to add a new instruction in the code, provided the raw bytes which represents the machine code.

`substitute_instruction_at()` Provided the instruction description pointer of the target instruction, it substitutes this one with the raw bytes passed as its argument.

Each new instruction added will require to update the entire set of relocation references in order to preserve the semantic consistency. This process is quiet expansive, since an instruction could references another one which is previous to it, then it is not possible to only update the instruction within the same function and successive to it. This will employ a huge time with respect to the other task, however it is done before the real execution, hence it does not hurt program's performance.

Reference update process starts by checking each instruction within the IBR independently of its parent function. It performs the following steps:

1. Shift instruction addresses beyond the one instrumented, through all functions

2. Check and update jump instruction displacements

   (a) In case of *short jumps*, Hijacker tries to use the original instruction. If the displacement is oversized, then the short jump will be substituted by a *long jump*.

3. Check and relocate symbols that references the instrumented instruction

Altering the chain by adding or even substituting one instruction, will require to realign addresses according with the shift amount due to the newly added instruction size. Therefore in the first step, the instrumentation engine will performs an update of each instruction beyond the one instrumented (replaced) by shifting them by the new size or the difference between the old one and the new substituted. As a minor optimization only the instruction beyond the one instrumented are checked. The IBR is structured in such a way to replicate the ELF linear structure of the text section, it is thus possible to update only successive instruction without hurting the correctness. Despite this optimization, the cost in time is still linear with respect to the input size, namely the number of instruction held by the IBR.

Herein, although instructions have the correct new address placeholder within the IBR, cross-references among them are not more valid since even relative displacement might be no longer aligned to each other. Due to the bytes added or removed in between, the second step is mandatory to keep code consistency by looking for each jump instruction to update its displacement. The update process should be straightforward, however it hides a challenge. Compilers, obviously, try to optimize as much as possible the executable file size, short jump instructions are therefore preferred whenever applicable. Nevertheless an arbitrary amount of bytes can be altered between two instructions. By altering the binary structure, offsets may likely become oversized with respect to the original jump's size. In such a case, the short jump must be upgraded to a long jump. A nested instrumentation loop takes place by transparently invoking the `substitute_instruction_at()` instrumentation function. It apparently seems to lead into a possible infinite loop, but it does not, indeed. Let us consider the case of four short jump instructions placed at the maximum distance, to each other, allowed by their offset size, and suppose to adds a generic instruction between the first and the second short jump operations. Consequently to the instrumentation action, second jump instruction shifts down causing the first one to be substituted with a long jump. By undertaking both of the previous actions, last two jumps will subsequently shift by the same amount of bytes, without changing the relative distance between them, and thus avoiding to issue any other nested substitutions. Again the time cost is linear with the number of the instructions registered.

The final update step will look for the other relocation references which apply to the `.text` section. The engine will look for the symbol `.text` within the IBR and recompute relative displacement held by the `addend` field. Time cost is function of the number of symbols held by the representation.

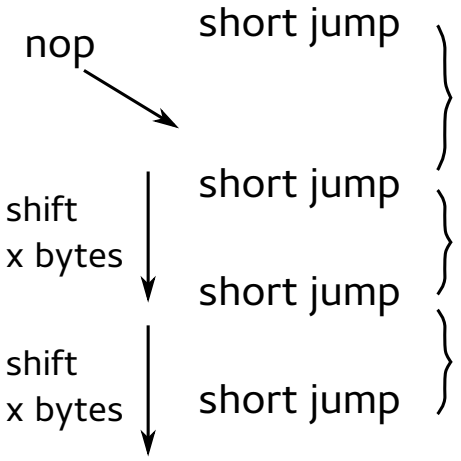


FIGURE 3.9: *Nested instruction instrumentation*

3.3.1 XML configuration rule file

Hijacker's back-end is driven by a simple xml file containing the instrumentation rules. Each instruction comes with a set of attributes that describes its scope, the instrumentation target, the number of passages, and so on and so forth. Each rule is provided as a xml tag, further illustrated in the Table 3.7 Table 3.8 eloquently exposes possible attributes for each entry along with a brief description. In such a way the user can widely customize the instrumentation engine according to its specific needs. With Hijacker we left open the opportunity to manually provide custom assembly code to inject, through the `<Inject>` tag. In such a way, the user is allowed to best refine the instrumentation; though it requires that he/she is informed about the specific machine to work with, and able to provide a compliant code. Injecting blocks of assembly instructions is a double edged sword: on the one hand it is a very powerful tool, but on the other hand, it may strain the user-side with a considerable effort. To cope with this issue, Hijacker provides a set of ready-to-use pre-compiled modules realizing the most common inspects, so that even less expert users can approach to it. The *monitor* module for reverse execution extension is one of those modules (refer to Section 4.2).

The configuration file is structured in such a way to progressively refine the action field. The user can exert a very fine-grained control over functions and instructions, by asserting which target have to be instrumented and how they must be altered.

Tag	Description	Scope
<Inject>	Allows to inject directly assembly code which should be written in the target machine specific instruction set, or compliant with some compiler installed so that Hijacker can compile it for you	
<Function>	Specifies a target function to which apply some instrumentation operations	
<Instruction>	Allows to specify a target type of instructions to be instrumented	
<AddCall>	Adds a new instruction whenever the aforementioned condition are met	

TABLE 3.7: *XML rule's tags description*

Scope Tag	Attribute	Description
Executable	reverseDebug	Enable the reversing module to produce revertible code on-the-fly
	entryPoint	Declare the executable's entry point
Function	maxPasses	Instruct the engine to instrument the current entity no more than x times
	exactPasses	Instruct the engine to instrument the current entity exactly x times
Instruction	injectAfter	
	injectBefore	
	replace	
	instruction	The family flag (see Table 3.4) of the kind of instruction to instrument
AddCall	where	
	function	
	arguments	
	convention	
	depthCall	
Range	callRepeatRule	

TABLE 3.8: *XML rule's tags description*

CHAPTER 4

The reversibility architecture

Speculative simulation, requires lots of rollbacks in order to undo out-of-order events. So far the most common and consolidated way was to employ checkpointing techniques, which though have large overhead either in time and in memory. Other researches in this field attempted to statically or dynamically reverse the code instruction by instruction, independently of the machine architecture. However, it turns out to be a challenging task. As previously hinted, it foremost necessary to figure out which logic is beneath a group of assembly instructions. Devising an architecture capable to effectively retrieve semantic from low-level machine information, such the one provided by object files, is not straightforward. In many works tackling this issue, this task is accomplished by first computing the control flow graph of the executable. It therefore allows to infer which logic result it has been computed. Nevertheless, as different human beings have their own writing style, so compilers have different ways to produce semantically equivalent code blocks, namely *idioms*. Henceforth for illustrative purposes, let consider Listing 7.3 and its relative assembly code (Listing 4), produced by GCC-4.9.2 compiler, which is reported in the Chapter 7 "Appendix D" for the sake of brevity. Lines 21–29 show an example of a compiler idiom which embodies an integer division.

```
1  int foo(int num) {
2    int x;
3
4    x = 10;
5    x *= 2;
6    x++;
7    x = x >> 2;
8    x /= 3;
9
10   x--;
11   x *= 3;
12   x = num + 8;
13   x %= 2;
14
15   return x;
16 }
17
18 int main(){
19  foo(5);
20  return 0;
21 }
```

The first approach we attempted is to fine-grain reverse each assembly instruction during the static binary analysis. By following this plain approach of reverse code generation, one could invert step by step each instruction met. However, this would turn into a misbehaving code whose semantic would be likely different to the original one. Hereafter, we refer to *instruction* as a single assembly statement, whereas *operation* is set of instructions which embody a logic calculation on data.

Let us consider the Listing 4 representing the set of assembly operations that realize integer division at line 8 of Listing 4. Without the knowledge of the high-level operation, namely `x /= 3`, we would have to treat a generic assembly block of code of which we even ignore the extents.

But let suppose to solve this first and non-trivial hurdle, the subsequent challenge would be to invert the following assembly block:

```
1 8b 4d fc              mov    -0x4(%rbp),%ecx
2 ba 56 55 55 55        mov    $0x55555556,%edx
3 89 c8                 mov    %ecx,%eax
4 f7 ea                 imul   %edx
5 89 c8                 mov    %ecx,%eax
6 c1 f8 1f              sar    $0x1f,%eax
7 29 c2                 sub    %eax,%edx
8 89 d0                 mov    %edx,%eax
9 89 45 fc              mov    %eax,-0x4(%rbp)
```

If we simply reverse each instruction from the bottom to the top we would obtain a completely different behavior. Table 4.1 traces registers' value change according to each assembly instruction. Lets suppose, for the example, that `x` variable holds a value of `5`. The above example clearly

Step	Instruction	RAX	RCX	RDX	Description
1	`mov -0x4(%rbp),%ecx`	?	0x5	?	Move the value in x into register RCX.
2	`mov $0x55555556,%edx`			0x55555556	Move constant 0x55555556 in register RDX.
3	`mov %ecx,%eax`	0x5			Copy RCX's value into RAX.
4	`imul %edx`	0x0xaaaaaaae		0x1	Multipy RAX by RDX, and store the result into RDX:RAX (RDX is sign extension).
5	`mov %ecx,%eax`	0x5			Move RCX in RAX.
6	`sar $0x1f,%eax`	0x0			Shift right by 31.
7	`sub %eax,%edx`				Subtract RDX from RAX.
8	`mov %edx,%eax`	0x1			Move RDX in RAX.
9	`mov %eax,-0x4(%rbp)`				Store the result back into x.

TABLE 4.1: *Assembly instructions' steps of the integer division by 5*

enlightens that is ambiguous how to handle the reversion process. Foremost, one can be observed that GCC does not employed the `idiv` division instruction to perform the operation, rather a multiplication surrounded by a set of seemingly unrelated instructions. The constant value of `0x55555556` used in step 4 to perform the integer division is the result of a trivial mathematical optimization [**?**, **?**], firstly proposed in [**?**] by Granlund and Montgomery which allow to achieve a speedup up to 4x. A more detailed digression is given in Chapter 7 "Appendix B". From the example would result clear that the instructions sequence are functional to perform the optimized integer division, however it is far from being self-explainable. Further, the operation performing likely disposes some information, which prevent the instruction reversibility. In such a case there is no other way to achieve reversibility but to save the intermediate results. The above integer division employees a shift operation which is a destructive instruction in nature: Hence, without knowledge of the logic result behind the sequence, there are roughly two mixable operating ways: (*a*) building a control flow graph through a static (or even dynamic) complex analysis, or (*b*) saving partial result so as they could be restored in future.

The knowledge of the semantic result is fundamental in order to properly reverse the forward code, otherwise some information might be lost. However this must not to be confused with

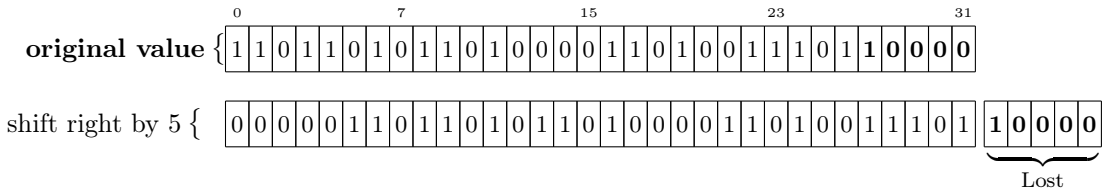


FIGURE 4.1: *Information lost through shift instruction*

constructiveness property of logic instructions. Instruction's constructiveness is a perpendicular —though related— aspect with respect to the depicted problem, since it regards correlation among instructions. Indeed a constructive operation could result in a destructive assembly idiom. Lets consider the Listing 4.1 which embodies the `x *= 3` operation, and suppose `x` variable holds a value of 2, at the beginning. Table 4.2 describes each computational steps and shows registers value changes. The operation involved is constructive in nature, but the instructions sequence is not.

```
1 8b 55 fc              mov    -0x4(%rbp),%edx
2 89 d0                 mov    %edx,%eax
3 01 c0                 add    %eax,%eax
4 01 d0                 add    %edx,%eax
5 89 45 fc              mov    %eax,-0x4(%rbp)
```

LISTING 4.1: *An example of a descructive operation's idiom*

Step	Instruction	RAX	RDX	Description
1	`mov -0x4(%rbp),%edx`	?	0x2	Load x value into RDX.
2	`mov %edx,%eax`	0x2		Copy RDX into RAX.
3	`add %eax,%eax`	0x4		Double the initial value.
4	`add %edx,%eax`	0x6		Add once more the initial value, so that to realize the third addition.
5	`mov %eax,-0x4(%rbp)`			Store back to x the new value.

TABLE 4.2: *Assembly instructions' steps of the multiplication by 3*

Most of the work done in other papers, such as in [**?**, **?**], is targeted to find a suitable way to efficiently reverse the whole assembly code generating a logically reverse clone that would be able to undo exactly what the forward code does. A burdensome effort is required to understand code instructions sequences and to build a control flow graph able to represent the logical program's semantic. Nevertheless, handling so much information introduces a considerable effort and could incur in heavily overheads, even though some interesting results were achieved.

Unlike the aforesaid paradigm, our approach is not aimed at producing a perfect logically reverse binary clone of the input. Rather, Hijacker relies on a novel technique which much simplifies the problem solution and the computational overhead required to achieve reversibility. A *MOV-oriented* reversing approach. From an high-level viewpoint, it represents a hybrid form which mixes together either reversibility and checkpointing paradigm. The following section will deeply deal about how it is implemented.

4.1 The reversing code approach

The idea beneath this approach tackles the unavoidable requirement to save disposed data packing it into an instruction rather than a memory location. We therefore blend together state

saving mechanism with instruction reversibility. Backward rerun instructions will, in fact, restore previous data.

Bearing in mind target applications, we observe as the only instructions relevant to our objective are those which may destructively alter the program's state[1]. Creating a detailed instruction-step reverse map could be functional to recover register values as well, which would not be worthy in the present case. On the contrary, the relevant element to our scopes is the net data flow towards memory, which directly translates to focusing on write instruction only. In particular, we currently operate on write `MOV` instructions. This represents a good trade-off that allows to achieve reversibility without the burdensome architectural complexity to build up, neither the need to statically parse executable's control flow graph. On the one hand, generating the inverse of a `MOV` instruction is quite linear, beside not straightforward. In fact, this process requires runtime information that relocatable files do not provide. It has to borne in mind that most of code references, values and parameters could be known only at link-time or runtime, thereby it is not possible to straightway reverse from the native code ignoring the underneath semantic. Basically, a `MOV` instruction will write a known amount of bytes to a runtime memory address overwriting a previous value we need to store. Although the first information can be easily gathered from relocatable format, the successive two are available only at runtime. From this consideration follows the hybrid instrumenting approach which blends static instrumentation with on-the-fly reversion. The former is functional to properly adds an assembly preamble code functional to prepare the reverse code generator's call which, in turn, does the real job at runtime.

As hinted above, static instrumentation process will add the call to a *monitor* module just before each memory-write `MOV` instructions. During this stage, Hijacker extracts relevant known meta-data from the targeted instruction and stores them into the `info_entry` data structure which holds the following information[2]:

- Size of the memory write operation (in bytes)
- Flags describing the addressing mode
- Address offset to which to write

The `info_entry` is packed on the stack along with the module's `CALL` instruction, so that at runtime, the control passes to our monitor. Upon its invocation, the monitor firstly retrieve the value residing at the address provided as argument, therefore it builds a new `MOV` instruction according to the written data size. Table 4.3 describes `insn_entry`'s fields. Newly created instruction has built-in the value that will be overwritten by the forward instruction. Since a former `MOV` will always predominates the rollback history within specific semantic-window, a further optimization is to effectively reverse only those instruction whose target address was not referenced yet. In such a way it is possible to reduce the total number of the reversed instructions.

4.2 Monitor: a reversing code module

As an important part of this thesis, we have designed and implemented a reverse code generator module for Hijacker. In particular, we want to evaluate its impact in the simulation environment. The monitor, is a compact module which generate reverse instruction on-the-fly, though it is not to be confused with a dynamic instrumentation tool (see Chapter 3 "Reference instrumentation tool"). Herein we want to focus about the implementation of the monitor module.

Previous static instrumentation stage inserted a call to this module just before each destructive `MOV` instruction so that to preserve old values to be definitely overwritten. Inverse instructions

[1] Also the CPU's context belongs to the program's state, however we are interested to memory only, since the actual execution of one event is usually considered atomic in standard PDES.

[2] This is a "disassembly-information caching" which aims to enhance performance, as discussed further.

are stored straightway on a private heap region. This is an entry point to reversible functions that the forward code can access in any time.

4.2.1 Interpreting the instruction

During this stage, Hijacker instruments the native code by inserting an assembly block constituted by the following instructions:

1. `pushfw` Save the `FLAGS` register's status
2. `sub` Move the current stack pointer. It makes room in the stack for the subsequent `insn_entry` structure injected by mean of `mov`s instructions.
3. `mov` Push the `size` field
4. `mov` Pack and push fields `flags|base|idx|scale`.
5. `mov` Push 4 bytes of `offset` field.
6. `mov` Push 4 bytes of `offset` field.
7. `add` Restore the stack pointer. The transition must be transparent to the running program.
8. `popfw` Restore the `FLAGS` register's status

This assembly block is functional to call the monitor and to instruct it. Once invoked, the module will find instruction data on the stack. At runtime, the module needs to know (a) the address at which the instruction will write, and (b) the size of data to be written, in order properly generate the reverse `MOV`. Those two pieces of information are retrieved by the Hijacker's instrumentation engine that disassembles and inspects the instruction itself. Computed data are prepared for the call to the monitor module, so that needed arguments are correctly passed to it. Engine is developed under the *ABI System V* specification, therefore `rdi` and `rsi` registers are used to respectively pass to the module the address value and the write size. From now on, the module has all the necessary information in order to properly generate reverse instruction on-the-fly during runtime execution.

This module is logically constituted by two parts: (a) the inspecting monitor and (b) the code generator. Former block is written directly in assembly code, in order to achieve maximum efficiency and is pursued to retrieving instruction meta-data. The latter, is indeed the one in charge to runtime generate the code further stored into the program's heap. In the following we propose the monitor's code snippet relative to the x86 machine architecture only.

Field	Type	Description
`size`	`unsigned int`	How many bytes the operation will write.
`flags`	`char`	Which kind of addressing mode the `MOV` operation uses.
`base`	`char`	The base register (if present).
`idx`	`char`	The index register (if present).
`scala`	`char`	The scale register (if present).
`offset`	`unsigned long long`	The address offset, relatively to the instruction, to which the operation will write on.

TABLE 4.3: *Description* `insn_entry`*'s flags*

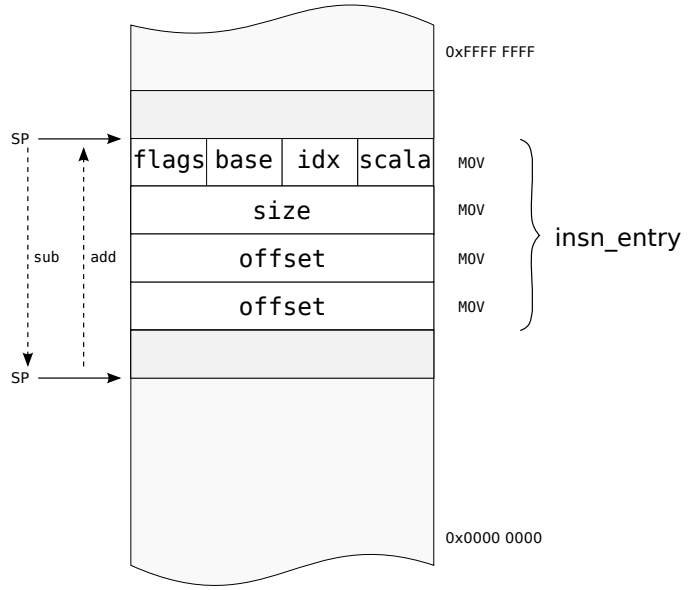


FIGURE 4.2: *Assembly instructions to push* `insn_entry` *on the stack*

```
1  monitor:
2    push  %rax
3    push  %rcx
4    push  %rdx
5    push  %rbx
6    mov %rsp, %rax
7    sub $8, %rsp
8    add $32+16, %rax
9    mov %rax, (%rsp)
10   push  %rbp
11   push  %rsi
12   push  %rdi
13   mov %rsp, %rbp
14   add $64-8, %rbp
15   lea 16(%rbp), %rdx
16   movsbq  4(%rdx), %rax
17
18 .PlainWrite:
19   xor %rdi, %rdi
20   testb $4, %al
21   jz  .NoIndex
22   movsbq  6(%rdx), %rcx
23   negq  %rcx
24   movq  (%rbp, %rcx, 8), %rdi
25   movsbq  7(%rdx), %rdi
26   imul  %rcx, %rdi
27
28 .NoIndex:
29   testb $2, %al
30   jz  .NoBase
31   movsbq 5(%rdx), %rcx
32   negq  %rcx
33   addq  (%rbp, %rcx, 8), %rdi
34
35 .NoBase:
36   add   8(%rdx), %rdi
37   movslq  (%rdx), %rsi
38
39 .CallDymelor:
40   cmp   %rdi, %rsp
41   jb  .End
42   call  reverse_code_generator
43
44 .End:
45   pop %rdi
46   pop %rsi
47   pop %rbp
48   add $8, %rsp
49   pop %rbx
50   pop %rdx
51   pop %rcx
52   pop %rax
53   ret
```

LISTING 4.2: *Inspecting monitor disassembly*

The Listing 4.2 depicts the real monitor module assembly code. First it performs a dump of all registers, therefore it computes the base address for the structure `info_entry` which holds current instrumented instruction's meta-data to retrieve. The `rdx` register is used to point the structure. Data pack is placed within the stack in the following order:

The structure is placed in the stack by preserving its natural flow. Therefore, whenever invoked at runtime, the monitor will displace from the base address in order to check which flags the instruction has, and accordingly to them it jumps to the proper code statement. From line number x-y, module rebuilds the offset by interpreting the displacement form used. In the x86, addressing mode are quite complex to understand, `operand := offset [base, scale,`

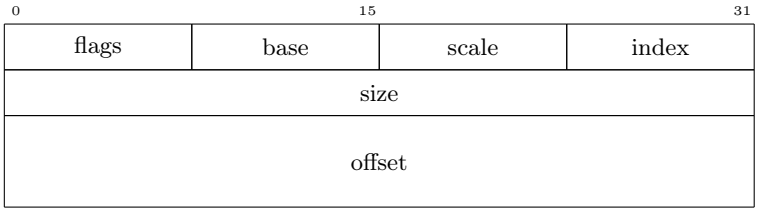


FIGURE 4.3: *The* `insn_entry`*'s bytes arrangement within the stack*

`index]`.

offset An immediate offset to adds to the remainder of the address

base An optional register holding the base value to which adds optional values

scale An optional register holding the value by which multiply the index

index An optional register holding an integer value

Machine microcode will compute the address as: $address = offset + (base + scale * index)$.

To subsequently call the higher-level part of the module, the assembly monitor preamble computes the destination address and retrieve how many bytes the instruction will write. Once this quest is completed, it checks whether the address falls outside the current stack window, otherwise no special action is undertaken. This is a duly optimization that leverages simulation application caveats. A further discussion is faced in Section 4.2.4 "Optimizations". If destination address is not within the process' stack, therefore the assembly preamble finally calls the `reverse_code_generator()` function, that represents the "second half" of the module.

4.2.2 Runtime code generation

Until now, the inspecting monitor has interpreted the instruction to be reversed and has properly built the call instructions sequence towards the complementary part of the module, the one in charge to effectively emit the reverse code. As previously mentioned, code generator will allocate a private region on the program's heap to dynamically store generated instructions. Namely the *reverse window*, described by the `revwin` structure.

Monitor's assembly fragment ends its execution by calling the `reverse_code_generator()` C function. Following the *ABI System V* calling convention, registers `rsi` and `rdi` keep, respectively, the first and second argument of the called function. Instructions at lines 36–37 in Listing 4.2 store the destination address and the amount of bytes to be written, respectively, in the aforementioned registers. The `reverse_code_generator()` function will perform the following three tasks:

- Checking whether the address is already referenced
- Retrieving parameters and checks to-be-written data size
- Computing the instruction's inverse

It retrieves the two parameters describing the actual address to which the subsequent instruction will write on, and the size in bytes of data ready to be written. This parameter is basically needed to correctly interpret and store the actual data present in memory. The function checks whether the target address is already referenced by invoking the `is_address_referenced()` function, in which case write instruction will be ignored since it is not straight relevant (see

discussion in Section 4.2.4 "Optimizations") to our objective. Indeed, any former `MOV` instruction involving the same address is predominant and will be the last restored, therefore it would be not worthy to reverse the following ones, too. Otherwise, if the address has not be referenced yet, `create_reverse_instruction()` function is called which finally builds a new instruction according to the write size.

4.2.3 Reverse code window

Simulation process is composed by many logical processes executing events in parallel, therefore the dynamic code generator is specifically designed to handle multiple instances of the *reverse window* structure, according to the current running epoch. Epochs are related to the notion of process' logical virtual time, and each event is associated to a distinct logical time value. The whole set of all those reverse windows represents the program's history. The reverse code is generated according to this schema ensuring each logical process to have its own private copy of the structure. Table 4.5 describes the structure used to keep track of multiple `revwin` structures. Figure 4.4 depicts a logical structure of this window. The reverse window is a descriptor for the

Field	Type	Description
`size`	`int`	The actual size of the reverse window.
`address`	`void *`	Address to which it resides.
`pointer`	`void *`	Pointer to the new actual free address.

TABLE 4.4: *Revwin structure's fields description*

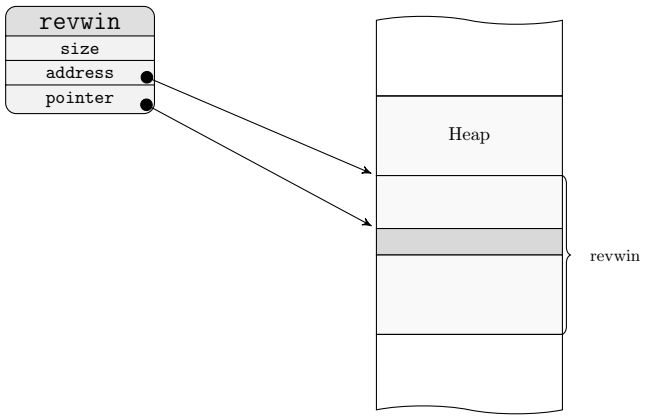


FIGURE 4.4: *Revwin descriptor*

meta-data needed to correctly handle on-the-fly code generation. Its structure is showed in the Figure 4.4. The structure resides directly into the running program's heap to allow a uniform and efficient way to invoke reverse functions whenever required. The allocation of a new reverse window is handled by the `allocate_reverse_window()` function. As default a `ret` instruction is added to the of the window; this is subsequently used during the restore process.

The reverse code generator uses the `mmap` standard library function to map and handle the revwin. To prevent security flaws, a heap window protection system is adopted. Generally, reverse window is kept under *write* permission only; upon a reverse execution has to be issued, a switch mechanism transparently grants a temporary *executable* permission to pages containing the reverse generated code. Suppose a `MOV` instruction is being reversed by our module while another execution thread would to invoke a reverse code block. Actually, this scenario would

lead into an inconsistency, since the new code might not be generated yet. Hence, to prevent critical sections, the protection mechanism is structured to grant either executable or writable permissions at same time. The `add_reverse_instruction()` function arbitrates the security mechanism by relying on the `remap` system call.

Field	Type	Description
`era`	`revwin *`	Holds an array of revwin.
`last_free`	`int`	Index of the last available slot.

TABLE 4.5: *Eras structure's fields description*

Each time a new instruction is inserted current `pointer`'s value is updated to the first available byte beyond the last reverse instruction stored. Since heap operations are much more restrained than ones operating on the stack would allow, the window size cannot be altered after the first initialization phase. This could impose a challenge in tuning the right allocation size for target simulation model.

Hashmap To allow a consistent integration of the reverse generator module within multi-threaded applications, all the data structures related to the handling of reverse windows have been declared relying on thread-local storage (TLS), which allows to transparently use different copies when a separate thread is being run by the original native application. This optimization has been specifically done considering that the final framework where we have integrated our solution is a multi-threaded simulation platform which explicitly benefits from this separation. While this, we emphasize that presented solution is much more secure in the general case, and that multiple revwins per each thread can be easily exploited by any application willing to rely on this module. As spatial locality teaches, most of the `MOV` instructions insist on the same address ranges. As hinted above, in case of a rollback the former instruction dominates latter changes. Simulation applications process events as "atomic" entities, if one event has to be discarded, it must be undone in its entire. Let us suppose to have a sequence of same-addressing `MOV` instructions history as in Figure 4.5, by proceeding backward each instruction will be progressively reversed until the event's beginning is reached. The last instruction to be restore is indeed the first emitted during natural execution flow. Thereby, generating two successive reversing instructions which alter the same address twice would be quite redundant and would introduce an extra overhead. To tackle this extra and unnecessary effort, we employ an *ad-hoc* structure to keep track of referenced addresses. This simple structure is a kind of hashmap described by the `addrmap` descriptor (Table 4.6). The hashmap is handled by `is_address_referenced()` function of the reverse generation module. Basically it exploits a two-leveled bitmap approach allowing to coalesce multiple addresses within a single long- or quad-word —namely 32 or 64 addresses, accordingly to the machine architecture—, so that to optimize space requirement for address mapping. A toggle bit is sufficient to indicate if an address is already referenced by some memory-write instruction or not. The structure is a linear array of elements treated as a bi-dimensional matrix. For this application, we assume a 64-bit architecture. Each element of the array is a quadword of 64 bits used as basic storage unit for a single range of family's addresses. 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

Field	Type	Description
`map`	`unsigned long long []`	Holds an array of range address entries

TABLE 4.6: *Hashmap structure's fields description*

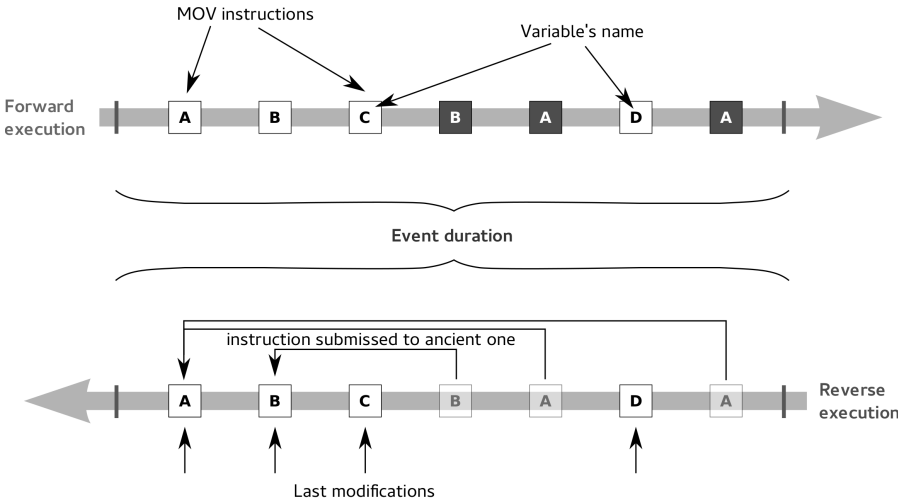


FIGURE 4.5: *Instruction predominance*

(i.e. the quadword). Aforementioned two indexes are computed by properly masking the address value. A family range is therefore composed by 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, ina 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 $n-1$ significant bits, while *index* is computed as the result of a bitwise-AND with the remainder of most significant bits. Figure 4.6 Shows an example of address' binding for a 32-bit architecture —we do not report 64-bit case for simplicity.

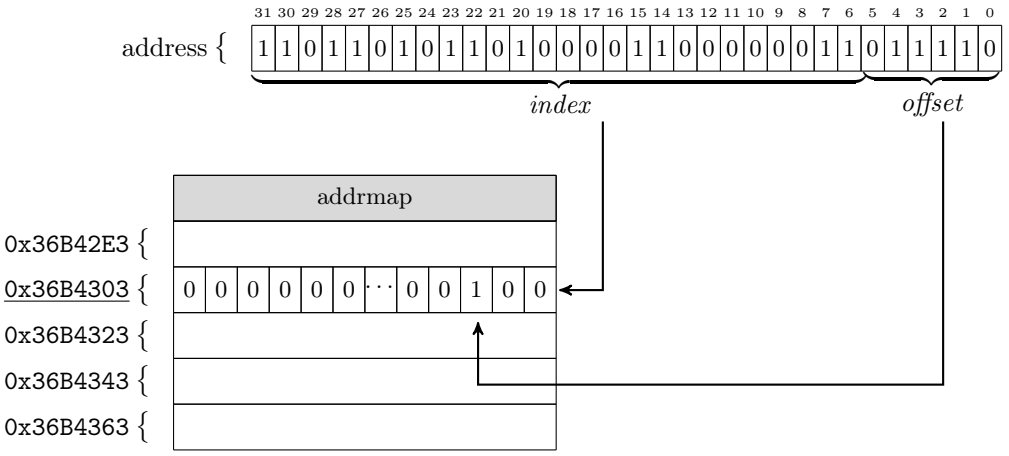


FIGURE 4.6: *The* `index` *and* `offset` *bitmasks of revwin's hashmap*

4.2.4 Optimizations

This section briefly describes implementation and constructive optimization guidelines we followed throughout the realization of this module.

First implementation of the *reverse code generator* was designed to heavily relying on malloc

external library which introduces a considerable latency due to the calling convention initialization phase. Conversely, to narrow as much as possible the runtime overhead inevitably introduced by the module, most of the memory operations can be simply solved by assignments. This noteworthy reduce the overall reaction time within a factor of 1.5x. The following Table 4.7 reports a first rough overhead evaluation in either the two cases, (a) first unoptimized version and (b) the second optimized one. The *writes coefficient* ranges in $[10, 0.2]$, where at lower values correspond an higher write density.

Writes coefficient	Unoptimized	Optimized
10	0.74	0.5
5	1.34	0.92
1.8	2.23	1.39
1	10.23	6.82
0.8	15.57	12.36
0.5	25.58	17014
0.2	160.54	134.63

TABLE 4.7: *Optimized version runtime comparison*

A further optimization is made at runtime by the monitor module itself. `MOV` instructions which accesses addresses within the current process' stack window are dropped, as previously hinted. Those instructions are not useful for rollback goals, since they affects only local runtime variables of the current process which belong to the actual in-processing event. By storing also stack-write `MOV`s we would increases overall memory requirement pointlessly. Simulation events are indeed treated as atomic entities, and in case a straggler message causes to rewind the simulation, no event will be rolled back partially. Every involved event will be restarted from scratch, relieving from the need to recover its execution context. Yet from the first static analysis, it is not possible to infer whether an instruction accesses the stack or not, since relocatable file cannot provide such runtime information. Therefore during simulation processing, upon a call to the reverse code generator, it retrieves the destination address and checks if it falls within the current process' stack window. If this is the case, it simply returns control back to the native code, ignoring the instruction. Described mechanism is basically oriented to memory optimization rather than to curtail time effort. Table 4.8 reports the percentage of stack-write `MOV` instructions with respects to the total number of instrumented `MOV`s, varying the event density factor[3]. As it results, for significantly complex runs, stack-write instructions account for about an half of the total, that translates into a considerable memory saving. Marginally, this optimization

Event density	Stack-write	Total instrumented instructions	Percentage
10	2833	3802	74.5%
100	302096	567541	53.2%
1000	8614631	16561626	52.0%

TABLE 4.8: *Percentage of stack-write instruction with respects to total instrumented*

could statistically give a restrained time speedup, if we consider that a limited number of extra instructions would be executed. Further, just to have an insight on the actual weight of the static instrumentation, Table 4.9 shows the number of instrumented memory-write instruction, for our simulation models, with regards to the whole number of `MOV`s in the binary.

[3]Event density factor is related to the simulation model used, see Section 5.1 "The Simulation Model and its Configuration" for more details on actual experiments.

Instruction	Value
Total instructions	1718
Instrumented instructions	34

TABLE 4.9: *Memory-write instruction instrumentation statistics*

4.3 Selective reversing instruction

The work presented in this thesis leverages some caveats of the specific target application, namely PDES. As previously discussed, in our case there is no need to undo instructions that will be undone anyway by ancestor ones. Instruction predominance is a property that holds only under certain conditions. Specifically, when the rollback has to be performed in its entirely to the beginning of its recording state. In those cases, it is straightforward that any leading `MOV` instructions which write on a memory location will predominate over newer ones on the same address. Nevertheless, this is not a general case, and different approach has to be adopted to cope with partial rollbacks. As a future optimization reverse operations, we will devise a selective technique to dynamically choose only the right instructions to be undone in order to reach wanted state.

Our approach generates a reverse code in such a way to allow conditional execution of the inverse instruction relying on the current timestamp information and the target one.

4.4 Parallel Simulation Platforms

Simulation is a problem-solving technique to cope with complex mathematical models generally conceived from real (or hypothetical) phenomena, which are otherwise not trivially reproducible. Let us consider few examples, such as determining physical constraints to build a bridge or a skyscraper, fluid dynamics simulation to realize airfoils or wind-sails, or rather in medical field to predict a viral diffusion, or again the description of a meteorological evolution for weather forecast, and so on and so forth. In many cases, real phenomena may not be empirically analyzed in nature, either due to their complexity[4], or because they are potentially dangerous. The only way to measure or assess the operating characteristics of a system is thus to observe it in actual operation. Simulation basically mimics real-world, therefore it provides best-effort approximate solutions; whether they are good or not depends on the knowledge and the cleverness of the model's creator. Computer simulation applications can be roughly described by the following perpendicular properties:

- Stochastic (e.g. Monte Carlo Simulation) vs. *Deterministic*
- Continuous vs. *Discrete*
- Local vs. *Distributed*

In particular, we are interested in the *Discrete Event simulation* (DES) which is defined by Nance in 1993 as follow:

[4]Many of the microscopic phenomena in fact cannot be analyzed due to insufficient information irreproducibility

> Discrete event simulation utilizes a mathematical/logical model of a physical system that portrays state changes at precise points in simulated time. Both the nature of the state change and the time at which the change occurs mandate precise description. Customers waiting for service, the management of parts inventory or military combat are typical domains of discrete event simulation.

A further evolution to cope with complex models that require considerable computational and storage requirements, is the *Parallel-DES* (PDES) which exploits the load distribution within clusters of different computing nodes. Parallel simulation can be broadly classified as (a) conservative and (b) speculative, according to the causal consistency property strength they ensure. *Conservative* simulation relies on locking strategies to avoid out-of-order events ever occur. On the other hand, *speculative* simulation exploits optimistic paradigm which eagerly tries to maximize underlying parallelism by relaxing message causality constraint.

Conservative technique adopts a locking mechanism on the scheduling queue until the simulation object is considered *safe* again. The object safety is determined by the causality order relation. It is evaluated by the *lookahead* value, which is the number of the LVT units that a generic LP_i can just "look ahead" in order to predict if a generic unprocessed event would hurt the causality order. By quoting [**?**]:

Definition 8. *(Lookahead) If a logical process with LVT of T can only schedule new events associated with timestamp of at least $T + L$, then L is referred to as the lookahead value for that process.*

Therefore in conservative approach, each queued event has to be processed in a non-decreasing order. If this property cannot be ensured, then the simulation object is considered *unsafe* and "locked" until the overall simulation process realigns to a further safe state. The larger is the lookahead value, the more will be the performance, since less locks are needed. On the contrary, if the lookahead value is too small, it generates a bottleneck. Tuning the lookahead is possible both statically and dynamically by automatic adjusting algorithms. Some techniques was developed to dynamically keep this parameter large enough, basing on events pre-computing techniques or on the *object distance*. This is a kind of lookahead metric value, computed from the time difference between LPs, such that two events can be reasonably considered unrelated.

Conservative approach ensures that no event will ever violate the causality property. However, this entails performance losses since parallelism would not be really exploited, indeed. By adopting a locking mechanism the system partially serializes the events processing, and according to the application characteristics, even if it is not strictly necessary. Let us suppose two logically unrelated events e_1, e_2 the system actually misjudges to predict; therefore e_1 and e_2 will be serialized, even though they could be processed in parallel. Further, this approach involves *deadlock* issues that necessitate to be properly treated. Locking the execution of simulation objects can lead to the mutual awaits for an unlocking event, which though cannot be delivered since no more LPs are actually active.

Contrarily to the conservative approach, the optimistic one is based on the speculative processing of all the scheduled events ignoring logical and causal relationship between them. In such a way, the system cannot ensure that events are actually performed in the causality order, but guarantees to exploits parallelism and whole computational resources. Since there is always the possibility that a *straggler* message comes up with a timestamp lower that the current LP's time. The system has to be silently recovered back to a coherent state. From a logical perspective, each LP locally rolls back all the events associated with a LVT value grater than the straggler's one. Further, for those events that have been caused, in turn, a remote message to be sent, the so-called *anti-message* has been generated and sent to the same destination. Anti-messages convey sufficient information to issue a rollback on the target process, too.

On the one hand the speculative approach guarantees higher performance by entirely exploit parallelism, and further to not incur in deadlock situations. Since it does not rely upon the lookahead value —which needs to be computed each time— to decide if to start a recovery procedure, it entails a simpler architecture. Synchronization among processes is transparent with regards to the final user and does not require much more effort to be implemented. On the other hand, it requires an optimal strategy to periodically save the sate for the recovery support. Speculative simulation, therefore, significantly burdens storage demand and introduces additional overheads due to the management of those subsystems.

As discussed in Section 1.3 "The Rollback Operation", most applications need to eventually rollback a faulty state to a consistent one. The common technique for realizing rollback is the state saving, whose naïve implementation stores the state's value whenever it is altered. Scope of this thesis is to assess the performance achievable by employing the reverse execution strategy as an alternative to pure state saving, and specifically targeted to our proposal.

4.4.1 Parallel Discrete Event Simulation

Generally, the simulation evolves by computing either continuous or discrete *events* which alter a set of memory items representing the *state* associated to the simulation.

Definition 9. *(Discrete event) An event is defined to be discrete if it occurs at precise point in time and its execution is impulsive. That is, it has no notion of time during its performing.*

The underneath idea of PDES is to partition the main simulation model into several sub-problems held by as many simulation objects. They interact with each other by exchanging information messages in the form of discrete events[5]. The whole simulation process has a hierarchical structure where the bottom level is represented by simulation objects, embodied by *Logical Processes* (LP) which are in charge to process events. LPs, in turn, are managed by a simulation kernel which runs each on distinct physical CPUs, according to the initial configuration. The simulation process has a *global state* associated with it, that thoroughly describe how the model is evolving. On the other hand, the generic LP_i modifies its own simulation's *local state*, S_i. Local states must be disjoint to each other, and such that they form a complete partition with respect to the global simulation state.

$$S_i \neq S_j \forall i, j \in N \tag{4.1}$$

$$\cup_{i=0}^{N} \{S_i\} = S_{global} \tag{4.2}$$

The notion of time in discrete simulation is still fundamental, though it is unrelated to real-world one. Time is marked by logical units according to simulation model's characteristics. Within each real-time unit can be processed several logical time ones. In the PDES, as the global state is partitioned among LPs, likewise each LP has its Local Virtual Time (LVT), which describes the causal order of the events delivery. During the processing of one event, other can be delivered at the same logical time due to the parallel nature of the application. Each event is thus marked with a timestamp grater or equal to the virtual time associated with the one currently in processing. Let e_k be the event generated during processing event e_j, the associated timestamp respectively t_k and t_j must ensure that: $t_k \geq t_j$. However due to the communication layer, messages could experience latency and not delivered consistently. It can be likely happen that two events e_x and e_y violate the causal property, meaning that $t_y \leq t_x$.

Definition 10. *(Causal consistency) Two events e_j and e_k delivered to process p_n, which respectively occur on time t_j and t_k, are causally consistent if they ensure the following relation:*

$$t_k \geq t_j \tag{4.3}$$

[5]From a logical point of view, PDES environment enforces *message* and *event* concepts' identity. Therefore, hereafter we use them interchangeably.

As previously hinted, two are the approaches to cope with causal consistency: (*a*) conservative and (*b*) speculative. We focus on the latter strategy though it allows messages to possibly violate the property —the so called *straggler messages*. This will happen upon the delivery of an event e_s with a timestamp t_s lower than a t_p associated with an already processed event e_p. In such a case, the application will leads into an incoherent state, which requires a rollback process to be undertaken in order to restore the system. Events with a timestamp $t \leq t_s$ are no more causally consistent and must be undone (Figure 4.7).

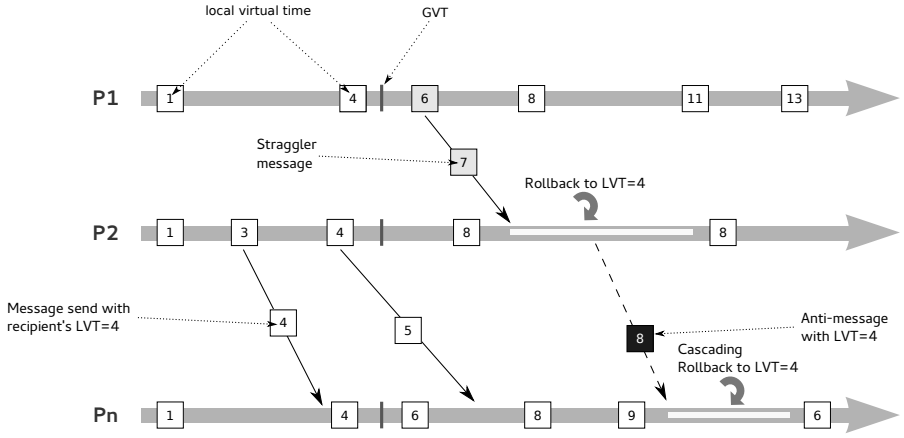


FIGURE 4.7: *Example of rollback due to a straggler message*

Events in the aforementioned logical time interval represents the so called *rollback distance* which can be arbitrarily high, according to the relationship among events. To avoid to rewind an excessive —and foremost unnecessary— number of messages, Jefferson in [**?**] proposed to adopt a Global Virtual Time (GVT) which corresponds to the lowest common LVT among all the LPs. The GVT represents a sort of committing frontier, before that events will not be rolled back anymore. The GVT subsystem is in charge to periodically compute a new common minimum event's timestamp among the LPs' queues. If a new GVT value is actually available, it will be notified to all kernels, that can issue a *fossil collection* process. All the memory buffers associated with previous events, with respect to the new GVT, can be securely released, optimizing the overall storage requirement.

$$GVT = \min\{t_k | e_k \in UnprocessedEvent\} \qquad \forall k \in N \tag{4.4}$$

Where N is the set of all the LPs, and LVT_k is the virtual time associated to the LP_k.

Whenever a rollback takes place, (*a*) the current processing event must be discarded and (*b*) a previous snapshot must be restored. The second point is the quite thorny, since it challenges either the program performance and the application transparency. Programmers, indeed, should not to be aware of the rollback mechanism whose implementation should hide its complexity away. On the contrary, in this thesis we are interested in a more attractive solution embodied by the reverse execution. One of the first approach was devised by Carothers, Perumalla and Fujimoto in [**?**], where they address the issue in compliance with the Time Warp protocol [**?**].

Suppose two active LPs, LP_1 and LP_2, running at a certain time distance each other and such that LP_1 is running faster than its mate. Therefore the probability that a straggler message is delivered to LP_2 form the former process is proportional to the temporal distance between the two. If the straggler message does not cause a cascading rollback, only a local rollback will be issued by the simulation kernel of the latter process. However in this scenario distance between LP_1 and LP_2 will increase due to the latency introduced by the local rollback on LP_2, which cause the time distance to grow again and thus the probability a new straggler message will

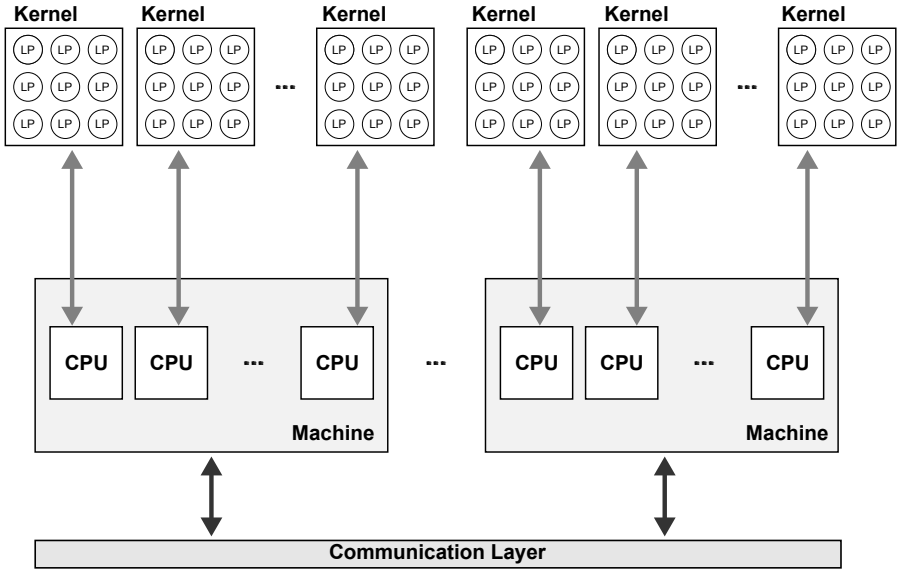


FIGURE 4.8: *PDES's architecture block diagram*

be delivered in the future. Even worst is that the higher the distance will be, the higher the probability that the straggler message would generate a cascading rollback. Parallel simulation is a non-deterministic process in nature. A rollback process will look for the nearest checkpoint in time such that all the messages before the straggler are undone. However since the snapshots are discrete in time, once a previous checkpoint is restore it is likely needed a *coasting forward* process; the forward re-execution of correctly processed events that have been collaterally undone. Once the system is realigned the simulation can proceeds normally. In order to properly rewind the overall state it is necessary to guarantee that non-deterministic operation, such as ones that relies on asynchronous inputs, or like random number generators, would execute in the same way as they previously do.

How CPU are assigned to the actual LP, is another challenging issues to cope with. As the figure 4.8 illustrates, each simulation kernel — which is statically assigned to each physical available CPU in the systems— hosts a set of logical processes which have to be properly scheduled. CPU-scheduler can considerably affect the simulation performance, therefore it has been chosen cleverly, according to the specific application purposes. Although there are some artistic implementation, the common choice is the Lowest Timestamp First (LTF), which selects the LP whose "next" event has the lowest timestamp within the set of ones managed by the local kernel.

4.5 The use case: ROOT-Sim

ROOT-Sim (The ROme OpTimistic Simulator) [**?**] is an open source simulation kernel developed as a static library which can be linked to any other executable[6] for general purpose simulations. This chapter shows the framework's architecture which is roughly composed by three layers; an application level, on which resides the final user's application represented by the simulation model. The top level relies onto the kernel mid-level, which represents the core set of subsystems in charge of managing the simulation process. Finally a communication layer allowing LPs to exchange messages. ROOT-Sim employees the MPI (Message Passing Interface) library [**?**, **?**]. Following the Time Warp protocol, ROOT-Sim enforces a logical identity between events and

[6]ROOT-Sim is developed in ANSI-C programming languages

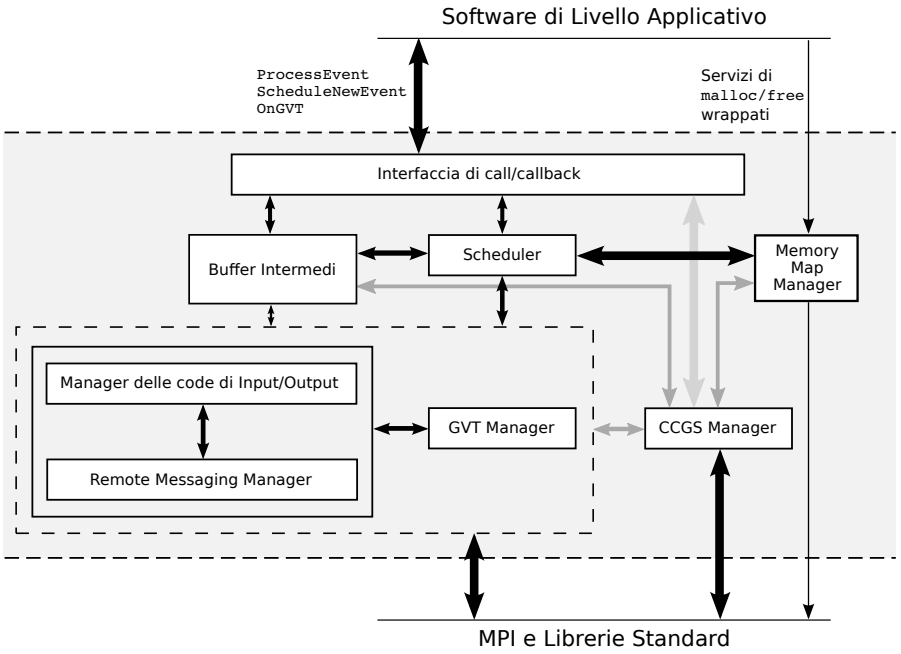


FIGURE 4.9: *ROOT-Sim's architecture block diagram*

messages, which convey the sufficient information to schedule a new event. Once the very first `INIT`[7] event is broadcast, the simulation proceeds through the LVT of each LP that serves incoming scheduled events. The choice of the scheduler is another thorny issue to cope with, since it can heavily affect the overall simulation efficiency.

4.5.1 API

Simulation model resides on the application layer, where the final user deploys his/her program. The user interacts with the ROOT-Sim library through a small set of functions provided by the library's API. Through them the application layer is allowed to communicate with the simulation kernel which is in charge of managing inner aspect of the simulation regarding the maintenance of the framework rather than the advancement of the process itself. Chapter 7 "Appendix A" reports an example based on the one provided in [**?**].

`ScheduleNewEvent()` Allows to generate and send a new simulation event to whichever LP. Destination kernel will undertakes the new message by scheduling it within the relative queue.

`ProcessEvent()` It is a callback function —which is mandatory to implement for the application in order to be compliant with the library— invoked by the simulation kernel. Actually it is the specific user implementation of the processing function performing whatsoever action the problem requires. It allows to deliver a scheduled event to the relative recipient LP, in order to be processed. `ProcessEvent()` will give back the control to the application layer which will processes the message. The causality order and the delivery is completely up to the kernel which is in charge either to keep events consistent —or transparently issues a rollback otherwise— and to choose the next message to deliver according to its typology. `ProcessEvent()` has no notion of parallelism, therefore the final user can implement it

[7]The INIT event is transparently scheduled to all the LPs, at simulation startup.

almost serially. Semantic coherence is guaranteed by rollback mechanism provided by the simulator itself.

`OnGVT()` Represents the second callback which is mandatory to implement. It is invoked upon the completion of a GVT phase reduction, which allows the simulation kernel to pass each LP a committed and consistent simulation state, which could be used either for inspection and generation of statistics during the simulation execution, and/or for (distributed) termination detection.

Name	Type	Description
`me`	`int`	The ID of the LP on which the event is scheduled
`now`	`time_type`	The timestamp of the local clock
`event`	`int`	Integer code describing the event type in the application context
`content`	`void *`	Pointer to the structure representing the application-dependent content of the event delivered
`size`	`int`	The size (in bytes) of the event's payload
`state`	`void *`	Pointer to the structure describing the application-dependent LP's sate

TABLE 4.10: *Parameters of function* `ProcessEvent()`

Name	Type	Description
`receiver`	`int`	The ID of the LP to which deliver the event
`now`	`time_type`	The logical time of the event at which the receiver must execute it
`event`	`int`	Integer code describing the event type in the application context
`content`	`void *`	Pointer to the structure representing the application-dependent content of the event delivered
`size`	`int`	The size (in bytes) of the event's payload

TABLE 4.11: *Parameters of function* `ScheduleNewEvent()`

Name	Type	Description
`snapshot`	`void *`	The last consistent simulation state, which can be used by the LP to decide whether the simulation can terminate or not
`gid`	`int`	The ID associated to the LP which is being scheduled for termination check

TABLE 4.12: *Parameters of function* `onGVT()`

4.5.2 Internal features

ROOT-Sim provides a set of configurable features, such as the GVT period which drives the fossil collection process. The narrower this interval is, the less memory requirement the system will need; though the simulation will experience a slower execution, since the collection will be issued more frequently.

As previously faced, since parallel simulation is intrinsically non-deterministic, ROOT-Sim provides a set of internal numerical library that wrap the original ones in order to guarantee the Piece Wise Determinism (PWD) paradigm. It is quite fundamental to correctly rollback

those events that carry non-deterministic operation, such as input-dependent ones or random-based, obtaining exactly the same output during the forward retrace. Otherwise, each restoration processes would hurt the semantic of the overall simulation; in other words unfeasible. Piece-wise-determinism paradigm is firstly proposed by Elnozahy in [**?**]. The internal library features the most common distribution functions:

`Random()` Returns a number in between [0,1], according to a Uniform Distribution.

`Expent()` Returns a random number according to an Exponential Distribution of mean value mean.

`Poisson()` This function returns the waiting time to the next event in a Poisson process of unit mean.

`Normal()` Returns a number according to a Normal Distribution with 0-mean.

`Gamma()` Returns a number according to a Gamma Distribution of Integer Order ia, i.e. a waiting time to the ia-th event in a Poisson process of unit mean.

`Zipf()` Zipf probability distribution.

Internal numerical library should be used instead of the standard one. During the initialization phase, a master seed has been chosen in order to set the remainder of the library. This seed can be either provided manually through the configuration file or picked randomly by the system itself. In fact, to ensure a correct rollback operation, the internal state of the random library should be restored as well. Since the standard numerical library is not aware of the rollback operation, this process would not be done, driving to an incorrect reprocessing of the simulation trajectory, e.g. in the case of the reprocessing of a set of events during the coasting forward phase. When a LP is rolled back, ROOT-Sim transparently restore the (per-LP) random seed, which therefore results in a transparent rollback of the random library as well.

As PDES framework, ROOT-Sim lays on a distributed protocol to initialize a new kernel on each machine configured. It is based on the aforementioned MPI library. Through the configuration file, the user can instruct the system about the available machine the can be used for the simulation process; therefore as many new simulation kernels will be created on each machine and assigned to its available physical CPUs. Finally each kernel is in charge to host an arbitrarily number of Logical Processes. During this initialization phase, ROOT-Sim tries to evenly distribute the spawned LPs among all available nodes, if the option *block* is provided or rather in a *circular* fashion, otherwise.

Like other parallel simulation frameworks, also ROOT-Sim adopts the common LTF scheduling strategy, though it also allows the user to choose on two different variants, (a) a linear scheduler good for relatively small simulations or (b) a probabilistic constant-time scheduler that is better suited for large set of LPs. Former linear scheduler reminisces the old Linux 2.4 stateless paradigm, which computes the *goodness* parameters for each issued operation. Whereas the latter is a more refined scheduler which keep a state representation of itself consistent along time.

Concerning the state saving method, ROOT-Sim supports the simulation's state scattering among memory segments and provides two operational strategies regarding logging operations. It supports either incremental or non-incremental technique (refer to Section 1.3 "The Rollback Operation"), and a further optimization that relies on the self-adaptive algorithms that interchanges between the aforementioned twos.

4.6 Integration with Reverse Execution

Our reverse code generation module has been integrated within the ROOT-Sim simulation kernel, following two steps. On the one hand, we have altered the final executable generation proper

of ROOT-Sim, adding one additional step which involves the actual invocation of Hijacker to instrument the application-level code. On the other hand, we have added some logic to the simulation kernel in order to execute the rollback operation adopting our hybrid approach.

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

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

We have altered this compilation process by inserting an additional step right after Step 1 in the previous list. In particular, during this additional step we explicitly call Hijacker, passing an ad-hoc configuration rule set. By the rules passed to Hijacker, all the `mov` instructions using as destination parameter any memory operand are instrumented, placing before them a `call` to the monitoring routine, along with several `push` instructions which allow to pass the monitoring module all the cached disassembly information required to reconstruct the target address, as described in Section 4.2.1.

This is the bridging point between the actual reverse generator module and ROOT-Sim, as the `revwin` data structures will be directly visible to the engine. As a note, we emphasize that calling Hijacker before linking the model with DyMeLoR does not pose any issue regarding memory management, as the reverse generation module relies on `mmap` to allocate memory to keep reverse instructions, which is not instrumented by DyMeLoR itself, and therefore allows to keep these buffers away from the LPs' simulation state.

The monitoring module exposes to any instrumented program the following C functions in order to interact with the reversing module:

`monitor_initialize()`
: This function initializes monitor's data structures for the very first use, therefore it is mandatory to call it at the beginning of the execution.

`increment_era()`
: This function increments the current era's counter. If the current era is grater than the last one when the module is called, a new reverse window will be created.

`free_last_revwin()`
: This function frees the last reverse window from the program's history.

Simulation objects perform events according to the local virtual time (see Section 4.4 "Parallel Simulation Platforms"), and each event lives within a unique LVT. The `eras` data structure enforces this property, allowing to build a local event's reverse history (see Table 4.5). The aforementioned structure has a fixed —yet configurable— number of slots available, representing pointers to `revwin` descriptors.

At the beginning of the simulation, we call `the monitor_initialize()` API function exposed by the reverse generation module, which initializes monitor's data structures for the very first use. Then, before the actual execution of one simulation event at a specific LP, we call the additional `increment_era()` API function. This call increments the current era's counter, so that upon the next invocation of the reverse generation module, a new reverse window will be created. Internally to the module, it allocates[8] a new `revwin` descriptor to handle a current event's reverse binary "snapshot". This allows ROOT-Sim to keep all the reverse instructions generated during the execution of one simulation event at a specific LP into a different `revwin`.

Once the execution of a simulation event is completed, the reverse code generation module ensures that in the current `revwin` all the reverse instructions are stored in the relevant buffer. We have therefore augmented the set of API offered by the reverse code generation module including the `get_last_era()` function, which allows to retrieve the id of the last used buffer to keep reverse instructions. This information is then stored by ROOT-Sim in a vector kept in the last taken snapshot.

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

1. Similarly to the traditional restore operation, the simulation engine scans the log queue in order to find a checkpoint C associated with a timestamp $T_c \leq T_e$;

2. The simulation engine then selects the checkpoint C_{next} associated with timestamp $T_{c_{next}}$ such that $T_c < T_{c_{next}} \leq T_e$, if any.

3. If this checkpoint does not exist, then it means that $T_c = T_e$ so a traditional restore is executed. If C_{next} is found, then the simulation engine computes how many events N_b should be executed in reverse mode (namely, the events between C_{next} and e) and how many events N_f should be executed in forward mode in *coasting forward* (namely the events in between C and e). If $N_f < N_b$, then traditional coasting forward is executed to realign the simulation state to a timestamp $T < T_e$. Otherwise, if $N_b \leq N_f$, reverse execution is selected.

4. Assuming that $N_b \leq N_f$, then the array in the simulation state is used to query the reverse code generation via the `get_era(int era)` API function, in order to retrieve the initial address of each set of instructions to invert a specific event.

5. Once the address of this buffer of instruction is retrieved, the execution directly jumps to the first instruction at that address, using a function pointer call. By the fact we end this buffer using a `ret` instruction, after one event is correctly undone via the execution of reverse instructions, the control is returned to the rollback algorithm.

6. Step 4-5 are repeated for every entry in the state snapshot vector keeping the eras.

This algorithmic steps allow to benefit from the reverse instructions generated during the forward execution of events, in order to reduce at most the time required to restore an event. Yet, a possible extension to this algorithm would be to estimate the times δ_f and δ_b associated with a forward or backward execution of the events, so as to switch between the two realignment actions using a more precise heuristic.

So far, the integration has dealt with the generation and usage of revwins to support reverse execution of simulation events. Nevertheless, this solution can still have a significant impact on memory usage as revwins are never released. To recovery memory, we have augmented the logic associated with the traditional fossil collection operation. Specifically, during the fossil collection,

[8] This is the only case in which a `malloc` system call is called.

the chain of logs is traversed in order to release memory buffers associated with logs which belong to a committed portion of the simulation trajectory. During the execution of fossil collection, whenever such a log is found, before releasing its memory we scan the vector of revwin ids, and we repeatedly call the `free_revwin(int id)` API function, which releases the the memory buffers previously `mmap`'ed to keep reverse instructions. In this way, we are able to recollect memory, which can be used again during forward execution to maintain reverse instructions related to the execution of additional simulation events.

CHAPTER 5

Experimental Assessment

This chapter is dedicated to the analysis of our overall approach, aiming at assessing the overhead due to the reverse code generator and at evaluating the benefits in the context of optimistic simulation, when running the rollback operation using the aforementioned mixed solution. This analysis will be carried out by using a real-world simulation application, modeling the behaviour of a GSM system.

5.1 The Simulation Model and its Configuration

As hinted above, our reverse code generator is tested against a mobile network model adhering to GSM technology, running on the ROOT-Sim environment (refer Section 4.5 "The use case: ROOT-Sim"). A GSM network is structured in cells which represent the access points for mobile devices, and whose coverage is modeled as an hexagon (Figure 5.1).

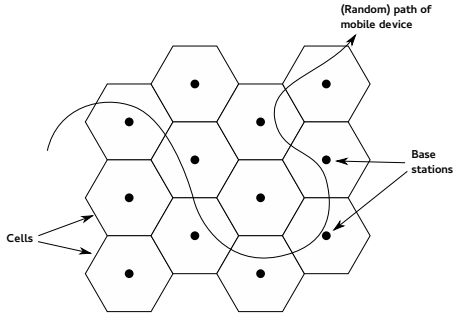


FIGURE 5.1: *GSM area network example*

Each LP models the state's evolution of an individual hexagonal cell, and the whole set of cells ensures wireless coverage on a variable size region. Cells handle a tunable number N of wireless channels, whose simulation models real power regulation and consumption taking into account attenuation factor and interference phenomena due to cell status and meteorological conditions, according to the results proposed in [**?**]. Each GSM cell keeps track of several attributes describing its current status, channel mapping, power consumption, registered devices and active calls. Upon the forwarding of a new call towards some device, the relative cell will catch it by instantiating a new record, therefore appended to the list of active calls. Whenever a call ends or the device moves out of its cell's range, it releases the corresponding record. In the latter case, the device will register to another cell which is in charge of handling the active call by instantiating the relative record. Upon call record creation, the active cell checks and regulates actual power transmission which involves a linear scan of the active calls list to compute the minimum value such that allows communication. Simultaneously to this scan, data structures holding fading

coefficients are updated according to meteorological conditions. Therefore, this scan entails an amount of memory-write operations which is directly proportional to the number of calls being currently active in a given cell.

Table 5.1 describes events handled by LPs during the simulation process, whereas Table 5.2 delineates configurable application's parameters.

Event	Description
`START_CALL`	Simulate a new call forwarding to a target cell
`END_CALL`	Simulate a call termination
`HANDOFF_LEAVE`	Model an active call transfer out from the current cell
`HANDOFF_RECEIVE`	Model the arrival of an active call from an adjacent cell
`RECOMPUTE_FADING`	Simulate a climatic variation which affects ongoing communications on current cell

TABLE 5.1: *PCS simulation model's allowed events*

Parameter	Description
N	The number of wireless channel
τ_A	The inter-arrival time of subsequent calls to any target cell
$\tau_{duration}$	The expected call duration
τ_{change}	The residual residence time of a mobile device into the current cell

TABLE 5.2: *Configurable PCS simulation model's parameters*

The *utilization factor* of available channels, expressed by Equation (5.1), highly impacts the granularity of the events.

$$utilization\ factor = \frac{\tau_{duration}}{\tau_A * N} \tag{5.1}$$

By increasing the channel occupancy, more power-management records are allocated which have to be scanned and updated, with a consequent grater latency in event processing and an increased number of activations of our reverse code generation module. Analogously, also the LP's memory requirement likewise increases, though it does not directly impacts reversing module's performance. If all channels are busy, any new call arrival is simply dropped, mimicking real-world scenarios.

To study the response of our proposal we use the following configurations of the PCS application. The foreseen call duration $\tau_{duration}$ is set to 120 seconds, while the residence time for a generic active call in the relative cell τ_{change} is set to 300 seconds. Further the inter-arrival time τ_A can vary throughout the process to simulate a load variation according to the period of the day, following the relation:

$$\tau_A = initial\ \tau_A \cdot DAY\_FACTOR \tag{5.2}$$

Where DAY_FACTOR represents the coefficient relative to a specific time slot; possible constant values allowed in the simulation application and its meaning are reported in Table 5.3.

Two different incarnations of this simulation model has been run. On the one hand, we have run a sequential model, build on top of an optimized $O(1)$ calendar queue scheduler. In this scenario, we have instrumented the simulation model's code using hijacker to silently insert the reverse code generator model, yet the reverse instruction are not used at all. This configuration, therefore, lies as a worst-case scenario for our final mixed rollback approach, as all the work done is never fruitful, mimicking e.g. a situation where all the rollback target simulation time

instants where a log is already available. Therefore, we are able to measure what is the overhead introduced by our approach. To have a more comprehensive view of the behaviour of the reverse code generator, we have varied the PCS' parameter *initial* τ_A in the interval $[0.1, 10]$ which, as mentioned before, produce a varied number of memory accesses and, in their turn, a different number of reverse code generator activations—smaller number of activations is actually associated with a higher value of τ_A. In this configuration, the number of active GSM cells has been set to 1024, each one managing up to 1000 wireless channels.

On the other hand, we have run a parallel simulation on top of ROOT-Sim, deployed on a 64-bit NUMA machine, namely an HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux Kernel version 2.6.32.5. The compiling and linking tools used are `gcc` 4.3.4 and binutils (`as` and `ld`) 2.20.0.

In this parallel simulation, we have used 32 Worker Threads in the ROOT-Sim kernel, each one bound to one CPU core of the underlying hardware machine. We have then run two different configurations of the benchmark, one simulating again 1024 cells, and another involving only 256. In both cases each cell was in charge of managing 1000 wireless channels. These two scenarios offer a different degree of parallelism, allowing to assess the behaviour when increasing the probability of rollback. In particular, the configuration involving 1024 cells has shown a rollback probability of about 3%, while the one with 256 has shown a rollback probability of about 15%.

Specifically, in these two contexts the application has been configured in order to simulate 17 hours of cellular system operation, from 00:00 AM to 17:00 PM. Therefore producing a τ_A variation in the interval $[0.64, 3.20]$, with one workload peak from the morning to lunch time, and one minimum load very early in the morning. According to previous variation, the utilization factor oscillates in the interval $[0.31, 0.06]$.

When running traditional state saving-based simulations, we have explicitly varied the checkpointing interval χ in order to compare the execution time of the reverse code generation-based simulation with different checkpointing dynamics. Specifically, the parameter χ has been varied in the interval $[1, 40]$

5.2 Experimental Results

In Figure 5.2 we show the execution time for the PCS model run on top of the sequential scheduler. In the plot, we compare the time required to complete a small number of calls (i.e., 1000 calls) when using the plain PCS model (referred to in the plot as *Non Instrumented*) and when instrumenting the model using Hijacker. By the results, we can see that running the instrumented version does produce an increase in the execution time.

Nevertheless, in Figure 5.3 we present the ratio between the intrumented and the non-

Constant	Value	Description
EARLY_MORNING_FACTOR	4	Simulates traffic density between 8:00 AM to 8:30 AM.
MORNING_FACTOR	0.8	Simulates traffic density between 8:00 AM to 8:30 AM.
LUNCH_FACTOR	2.5	Simulates traffic density between 8:30 AM to 13:00 PM.
AFTERNOON_FACTOR	2	Simulates traffic density between 13:00 PM to 15:00 PM.
EVENING_FACTOR	2.2	Simulates traffic density between 15:00 PM to 19:00 PM.
NIGHT_FACTOR	4.5	Simulates traffic density between 19:00 PM to 21:00 PM.
WEEKEND_FACTOR	5	Simulates the mean traffic density during weekend time.

TABLE 5.3: *Day factor values according to time slots*

	Component	Type	Amount
Memory	RAM	NUMA	64 GB
	Cache	L3	12 MB (4-core shared)
		L2	512 KB (private)
CPU	Processors		4
	Cores per processor	64-bit	8
	Total		32

TABLE 5.4: *Testing hardware's specifications*

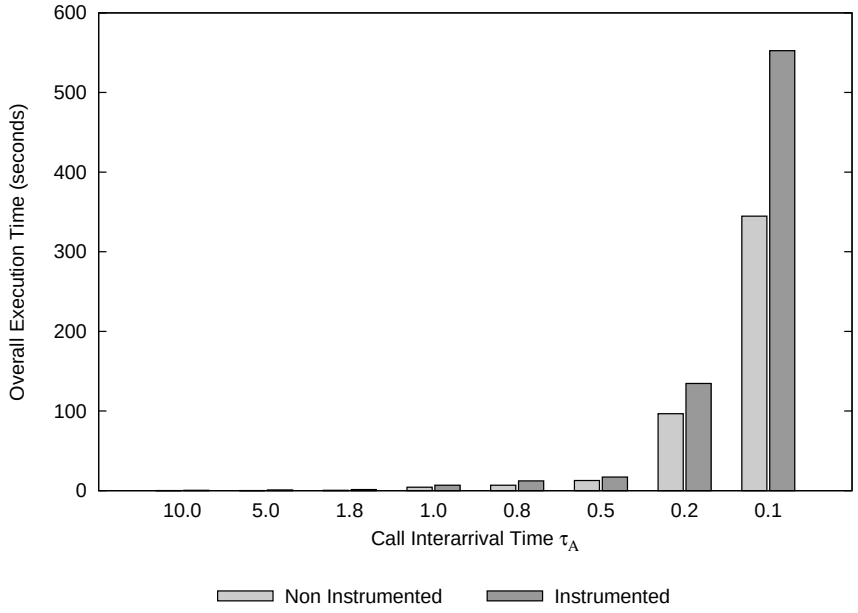


FIGURE 5.2: *Reverse code generator's time bare overhead*

instrumented execution time, in order to evaluate what is the actual slowdown introduced by our instrumenting architecture. By the results, we can see that in the worst case the slowdown is on the order of 5, but when the workload is increased (we therefore move from $\tau_A = 10$ towards $\tau_A = 0.1$, the effect of the overhead is reduced, giving a slowdown which is on the order of 1.5. This shows us that although the number of invocations to the reverse code generator increases, this effect is mitigated by the more complex nature of the events (related to the list scanning and fading recomputation, proper of the PCS application) and the overall cost is better amortized.

We have therefore selected $\tau_A = 0.8$ as the reference *initial* τ_A for the subsequent set of experiments, as it stands as good reference point for a configuration where the overhead due to the instrumentation is not enormous with respect to the actual global workload of the benchmark, thus making it interesting to evaluate what is the benefit of the mixed approach for state save/restore.

In particular, using this value of *initial* τ_A, we present in Figure 5.4 the execution time of the various parallel configuration when varying the checkpointing interval χ, the execution of the reverse generation-based simulation. We range this parameter in $[1, 40]$, where a value of 1 represents the basic copy state saving (refer Section 1.3.1 "State saving") technique, which takes a new snapshot at each event update. We additionally present the total execution time

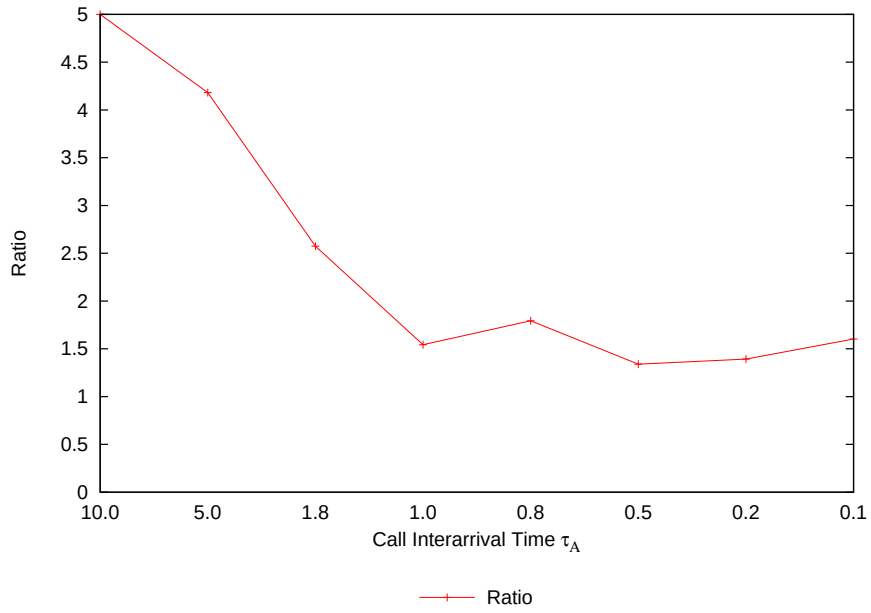


FIGURE 5.3: *Reverse code generator's time bare overhead*

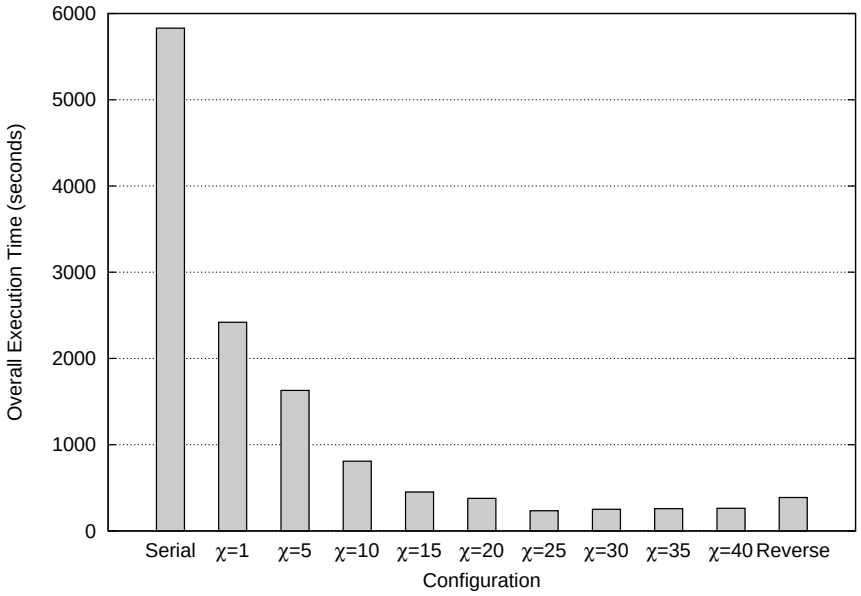
of the very same simulation model on top of the aforementioned constant-time calendar queue scheduler, yet without any kind of instrumentation of the application-level code. This, as reported in Figure 5.4a, confirms that our comparison is done among competitive parallel runs.

By the results in Figure 5.4 we can see that the best configuration, in terms of execution time, is the parallel one associated with $\chi = 25$. Nevertheless, the slowdown between this optimal configuration and the reverse code generation-based is on the order of 40%, while the speedup with respect to the worst configuration (with $\chi = 1$) is around 85%. Therefore, although not showing optimal, the reverse generation based simulation allows for a non-minimal speedup, e.g., in the case of non-piece wise deterministic simulations, where the user is forced to set $\chi = 1$ in order to obtain a correct simulation. Nonetheless, finding the optimum χ value is not often straightforward, it likely requires different tuning attempts or complex automatic algorithms that might not reach the objective as well. On the contrary, reversible rollback ensures more convenient way to proceed, and furthermore a more stable solution.

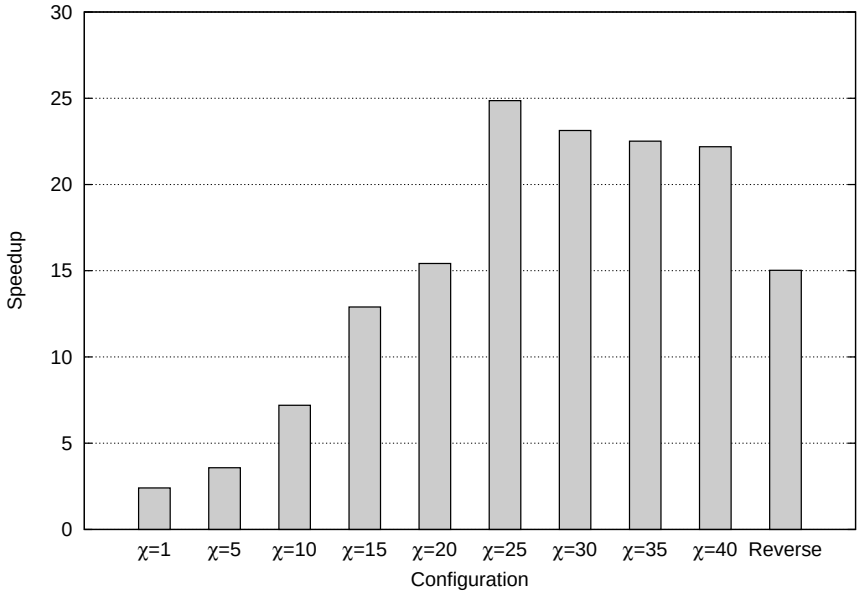
As event density increases, so does the likelihood straggler messages have been delivered and thus model gets more rollback-prone. By comparing the results in Figure 5.4 with the results in Figure 5.5, we can see that when the degree of parallelism is increased (namely, we move from 1024 hexagonal cells to 256, keeping the other parameters the same) the situation changes. In fact, while the reverse generation-based simulation is still not the best performing one, it shows a slowdown of only around 15% with respect to the optimal configuration found when $\chi = 5$. By the way, it follows grater memory consumption due to more frequent checkpoint interval.

Furthermore, the comparison between the results in Figure 5.4 and Figure 5.5 highlights that the reverse generation-based simulation is more resilient to variations of the optimal checkpointing χ_{OPT} in a given execution phase. This makes this approach more reliable if compared to other techniques which find the optimal checkpointing value χ_{OPT} relying on closed analytic formulas (see, e.g., [**?**]) which could nevertheless ignore secondary parameters which might play an important role in certain execution phases to correctly predict the execution dynamics.

As a final note, we report in Figure 5.6 the memory usage by several configuration of the ROOT-Sim platform when running the PCS benchmark with 1024 cells. Specifically, we have

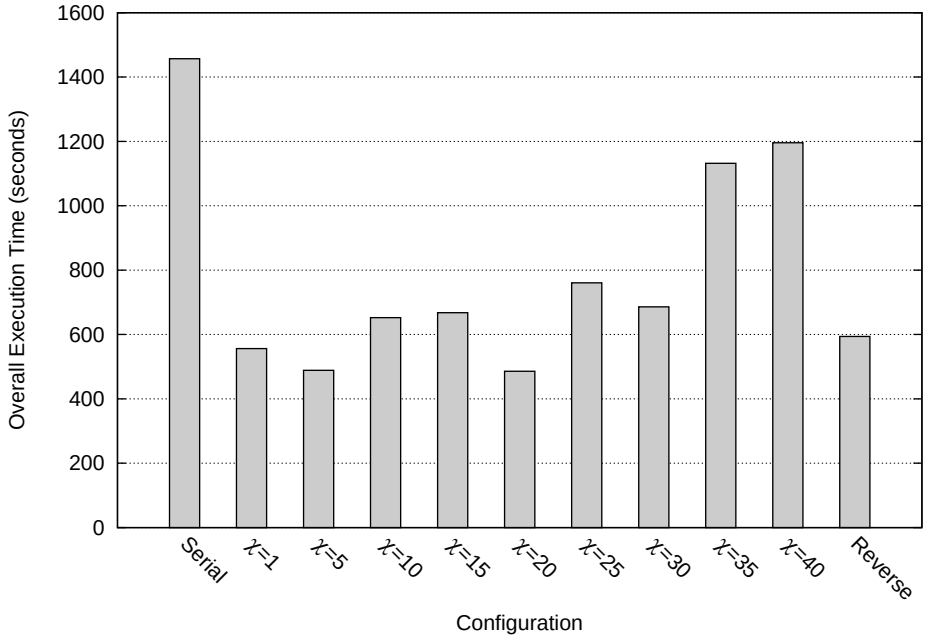


(A) *Execution time*

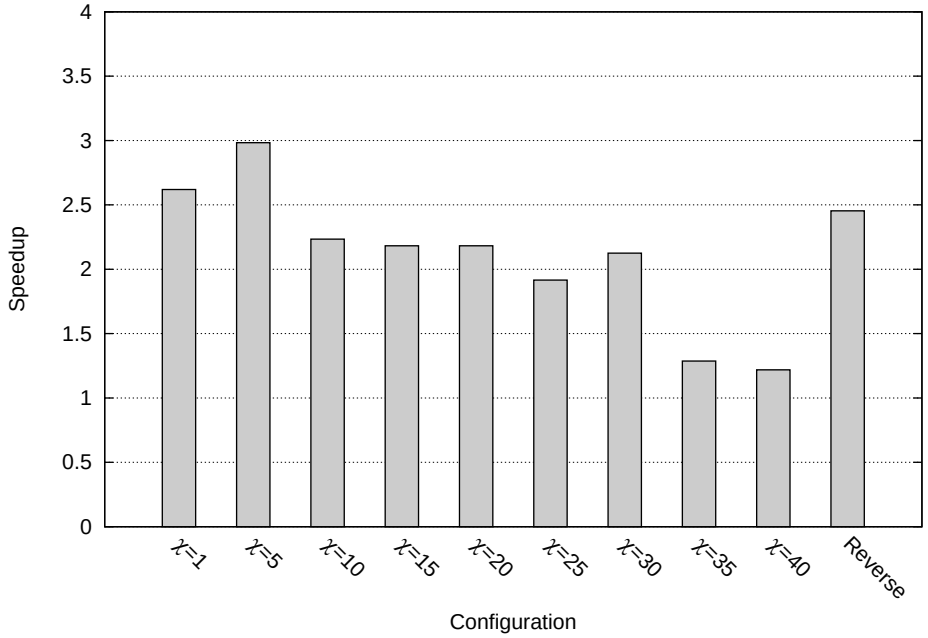


(B) *Speedup*

FIGURE 5.4: *Parallel Simulation using 1024 Cells*



(A) *Execution time*



(B) *Speedup*

FIGURE 5.5: *Parallel Simulation using 256 Cells*

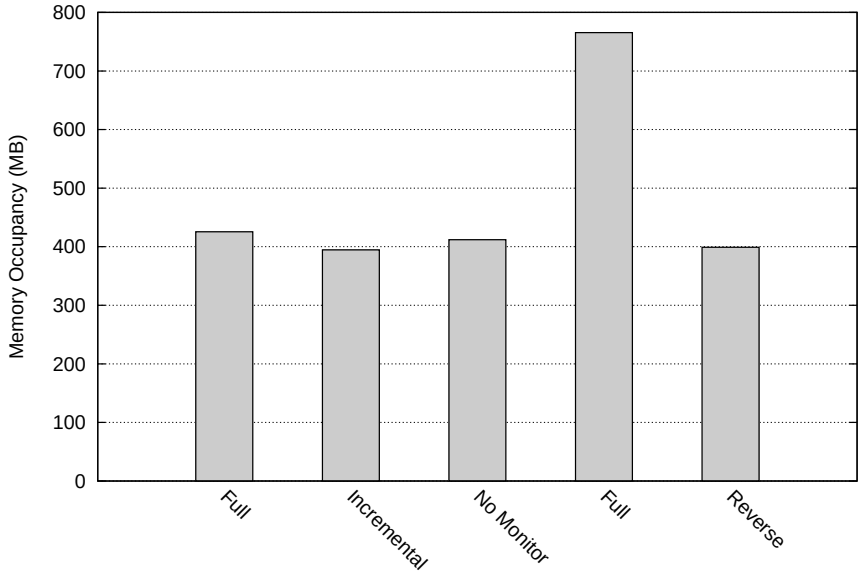


FIGURE 5.6: *Memory Consumption—1024 cells*

compared the memory footprint by the reverse code generation module only to store inverse instructions, with the memory footprint (both checkpoint data and metadata) by the various configurations of the traditional ROOT-Sim's memory manager [**?**]. By the results we can see that the memory footprint of the reverse-based solution is comparable to that of the incremental checkpointing, placing itself as a valid alternative (in terms of memory usage).

CHAPTER 6

Future application fields

This chapter is dedicated to future perspective outside the simulation environment. In particular we believe that our approach can be easily extended to other fields with a particular attention to debugging aiding tools.

Some architectures or systems do not allow an efficient debugging process to take place owing to an intrinsic structural complexity which thwarts canonical instruments to provide sufficient information to precisely determine the issue. Further, most of the test environments are virtualized or downscaled, with respect to the reality, to get developing and debugging processes cost-effective. For example, debugging ad-hoc system solutions may require to build a virtual execution environment to simulate them. Rather, parallel applications span over large clusters of powerful machines with hundreds of processors and concurrent threads, that could be very challenging to debug. Aside from the those bugs (e.g. due to communication exchanges) which can be easily solved during the early developing stages, deployed applications leave the programmer with a possibly huge data from which to fumble. To get out of such a fix, a solution is to scale down the problem on a more comfortable size; nevertheless, simulated environments do not always allow such a trick. Failure can be consequence of the input partition among clusters or processors which is not deterministic, or rather if the problem derives from rare conditions due only to large input scale. Much more bugs than would expected, remain hence "latent", due to an intrinsic irreproducibility; literature refers to it as the *bug reproducibility problem.*

Reverse Post-Mortem Debugging represents a new frontier not yet thoroughly explored, in this direction. Its application fields have a noteworthy impact in software development and moreover in code analysis. The approaches proposed allows to overthrow real time requirement (thereby money expenditure) by optimizing analysis' strategies, and providing a set of advanced tools which simplify developers' life.

Thus, taking into account such achievable time and economical benefits, why the reversible (post-mortem) debugging is not so widespread employed? The naïve answer to this question resides on the inadequacy of real implementations because of their complexity, but it is mainly due to the significant overhead brought by such an architecture. To gather enough information allowing to backtrace the execution, in the most general case debuggers might rely on pure state-saving techniques. Namely, it records each change within the program history so that it can be rerun in future. The aforesaid "record-replay" strategy is quite expensive due to the recording process, which considerably slows down the execution and further consumes much more memory to store the necessary data, with many similarities with the traditional state saving techniques in PDES where we have shown that our solution is feasibly applicable. So far, there are roughly two main ways to achieve reversibility:(a) by saving the execution history in its entirely or (b) through a reversible computation approach. The former is quite intuitive, whereas latter exploits a subtle intuition. Instead of recording memory states, the code itself may be reused to rewind the actual output as far as desired. Nevertheless, inverting code instructions is a non-trivial problem due to thorny syntactic and semantic language properties (see Chapter 3 "Reference instrumentation tool"). The overhead is basically composed of the time to either record memory changes or to

generate the inverse code, and the storage requirement itself. Needless to say that latter strategy might much more time efficient, though complex in its devise.

6.1 Reversible (post-mortem) debuggers

Reverse execution is commonly implemented relying on some state saving rollback&replay techniques. However, it exhibits a considerable storage amount which highly bounds performance and reversing capabilities, further it introduces a scalability problem.

Reversible debuggers allow to have a tight control on the program's execution. A very appealing solution for code debugging analysis, which is a tedious process requiring a considerable amount of time, spent on bugs disclosure. When a failure occurs, the very first action is the attempt to reproduce the bug, and generally, the source lines in the nearby of the failure firstly appears is the entry point where programmers start from. Most of the insidious bugs do not cause errors immediately, rather they likely cause the program to crash much later. Furthermore, bugs are much often weird, they hide themselves in nested loops or within complex interaction among software's modules or specific inputs, which may require an unpredictable number of iterations to reproduce them again. Traditional debugging methods are generally speculative-based and prompt the developer to choose where it is better to place breakpoints or print statements, and step forward the program's execution, even one loop per time, until the failure is reached. However, it is pretty common that even the most clever developer could overstep several times the bugs itself, without catching it. The higher is the complexity of the program, the greater is the probability to overstep the bug. Figure 6.1 clearly illustrates this time-consuming attempt. Cyclic debugging requires several rounds of program replays to finally reproduce the defect. Due to non-determinism further, the bug may likely do not befall again for several rounds, or even worst a different bug happens. As in physic, the observation of a process may be affect the process itself producing collateral faults unrelated to the original one, namely *Heisenbugs*. Analogously, different bugs could occur since the attempt to scale down the problem into a more comfortable size, which though alters inner interactions. Even though this approach is bearable for short running, it is unsustainable in parallel execution. Parallel applications (a) need much more time to run, and further (b) parallelism introduces a non-determinism issue[1].

6.1.1 Forward debugging

Traditional forward debuggers allow to only step the execution forward. This is the classical *cyclic debugging* approach, in which the failure is diagnosed by iteratively rerunning the program several times. It relies on the underneath idea to progressively gather data to zoom into the problem. To locate the bug, the user has no choice but to cyclically restart the program from the beginning, stepping through its execution and guessing whether to step over or into each function met. A single misjudgment leads in overstepping the bug causing to restart the program from the beginning, again. For very complex applications moreover, the failure can occur deep into; thereby the risk of spending too much time and to overstep the bug grows exponentially. It is therefore heavily funded on a user's guesswork. Trying to reach the point just before the bug occurs requires several re-executions of the program to progressively approach to the problem until it is got; a very painful and time-consuming work further affected to determinism problems.

To properly work, cyclic debugging requires that applications are fundamentally deterministic, such as non-interactive single-threaded programs. Parallel or distributed real-time applications, involve some kind of non-deterministic interaction, such as such asynchronous I/O or random functions, that cannot be reproduce each time in the same way. In fact, rerunning over and over

[1]Non-determinism is not a prerogative of parallel applications, there are lots of cases in which the debuggers itself affects failure occurrence by observing the state. That are the famous *Heisenbugs*

the program it does not ensure to succeeds in reproducing the problem itself or, rather, not to get into another one, as well. In those cases it is quite impossible to find the exact location of the bug without the risk of spending an eternity.

6.1.2 Reverse debugging

In contrast, reverse (or bidirectional) debuggers much shortens this awkward process allowing to step back to any prior statement just as the failure occurs, within the same execution context. Rather than trying to guess and to reproduce the error, it allows to diagnose just the failed run. Reverse debugging is an abstract generalization, but what are the real implementations? There are mainly few general approach, either (a) to log the whole program's history or (b) to rebuild the target state from some previous checkpoint or (c) to retrace backwards the same computational steps. The former solution represents the naïve and the simplest implementation, which in turn can be deeper differentiated into *trace-based* and *record-replay* strategy.

The program evolution checkpointing is a "middle-stage" solution. Whenever a bug befalls, the more recent snapshot is restored and the program re-executed up to the statement just before the crashing point. Since it is no more necessary to replay the program from scratch, a considerable part of the process can be skipped. However, it likewise introduces a time and memory overhead to compute and store, respectively, the snapshots (refer Section 1.3 "The Rollback Operation") during the forward run. Reverse debugging, on the contrary, overcomes the traditional error-prone process. It provides the most natural way to find and thus fix bugs, by allowing to naturally go back up straightway to the source. As human beings, we likely think in a reversible fashion as [**?**, **?**, **?**, **?**] show. Thereby would be quite natural, for us, to find the root causes of a problem by unwinding it in reverse. To quote from [**?**]:

> As a simple example, if you lost something valuable, you would go back to the places you visited during the day, one by one starting from the last place you visited. When a program goes wrong, it is tempting to do the same for debugging. One would wish to trace backwards an erroneous execution path to find the cause of unexpected behavior of a program.

Just upon a crash, the user can step back the execution flow until the misbehaving statement is reached in much less time and with much less effort.

To better understand the problem, we quote the following simple allegorical example of every day life, which properly catches process debugging difficulties and underneaths aspects of non-determinism.

> Train passengers have to take their seats. Either they have a seat reservation or not, in such a case those passengers randomly pick a free seat. Since passengers get aboard one at a time, those who have a reservation could find that the seat who they are entitled for is already occupied by another passenger, thereby they have to pick an available seat, too. Finally, it should not happen that some passenger X with a reservation remains standing up, since its seat is already taken and no other free ones are available. Such a case represents a *bug* in the system, thereby some action might be taken in order to find the passenger who had first stolen the seat.

This scenario, enlightens the power of reverse computation. As soon as a passenger remains without a seat it is possible to just backtrack the execution to find who peaked former's entitlement. Rather than trying to reproduce the issue in a separate run, which may lead into pretty different execution, one can fix the bug immediately.

6.2 Just a foresight

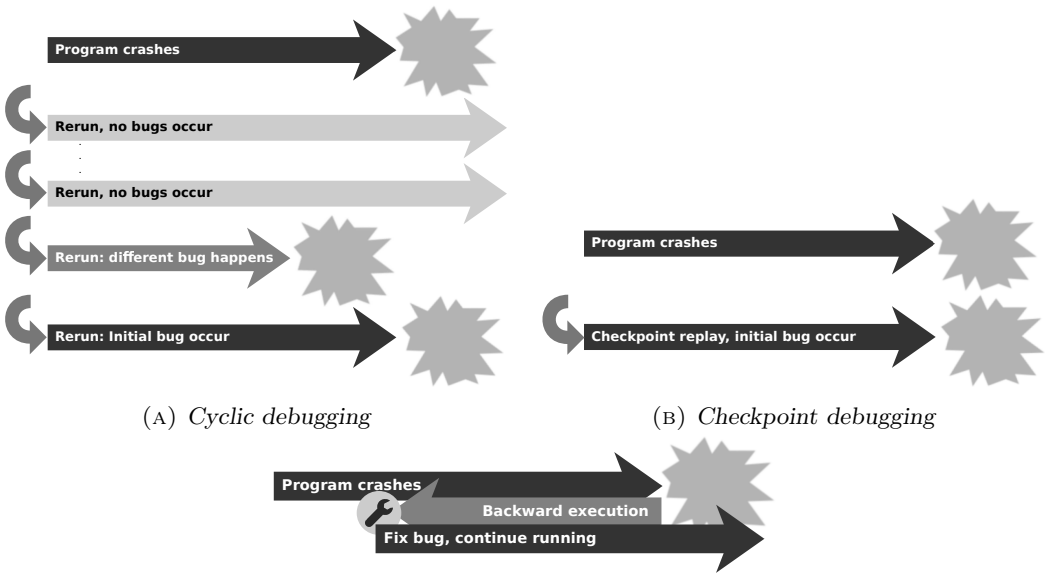
Hitherto given a brief overview on the debugging common techniques and challenges, this section is just an intuition on how the idea beneath our reversibility proposal could be exploited to achieve a more comprehensive notion of reversibility, namely "*revivability*".

Hijacker is an advanced instrumentation tool which aims to provide all the necessary supports to achieve execution reversibility and "revivability" through external debuggers. So far, we have seen how Hijacker parses an object file in order to build the internal binary representation which then has to be instrumented (Chapter 3 "Reference instrumentation tool"). Instrumentation allows to inject, or otherwise alter, purpose-specific code snippets to enhance debugger capabilities. Once Hijacker instrumented and rebuilt the object output file, it is possible to link it with the remainder of software's modules. According to provided instrumentation configuration, for example, the debugger can leverage those hooks to efficiently inspect the program in deeper detail.

Basically the idea is to provide sufficient information to whichever framework to achieve "revivability" of a crashed process, within the same execution context. Generally upon a crash, the operating system creates a core dump file which is no more than a memory map of the dead process' context. Nevertheless, since the very restricted amount of information provided, core dumps are just an hint for programmers during speculative breakpoint-based debugging process. On the contrary, the idea is to employ the reverse code generator so that at any time it would be possible to rely on a enhanced core dump. Following the same criterion used to emit and straightway push reverse instructions into process' heap, a single core dump file could become a sort of assembly program's executable footprint, retaining binary executable files. From the knowledge it would provide, clone context could be generated by backward running dump instruction. Once the program is again at runtime, the user as the possibility to freely step back and forth the same timeline at his/her will.

To achieve such an objective, the framework must have the following capabilities:

- It has to properly interpret crashed program's core dump, which can be straightforward if



(A) *Cyclic debugging* (B) *Checkpoint debugging*

(C) *Reverse debugging*

FIGURE 6.1: *Comparison between debugging techniques*

we consider Hijacker is already able to handle ELF binary files. This would represent a quite linear extension to what already implemented.

- It must have notion of where Hijacker's reverse modules have been stored, actually, the reversibility information within memory.
- Finally, it has to provide some user-friendly mean, such as an interactive shell, that allows to control program execution flow in both directions.

Basically, the work's main difficulty is to tackle arguments far away from each other. They span over different aspects of the information technology in its entire, from the lowest binary level up to the human interaction embodied by the final user that is performing the debugging activity. In particular, each of the above listed subjects introduce several challenges that require ad-hoc solution to be conceived, which can be successfully integrated for general-purpose software.

CHAPTER 7

Conclusions

In the this thesis we presented a fast and solid support to reversibility for the parallel optimistic simulation, as alternative to state-saving rollback. We have devised a reversible module for the High Performance Computing environment which dynamically emits reverse code straightway on the program's heap accordingly to an optimized criterion to filters hurting instructions.

We have achieved our purposes through an hybrid instrumentation approach of relocatable binary files. At compile-time the instrumentation engine injects few instructions that are functional to call our reverse code generator module just before the altering instruction is performed. Once invoked, it has all the information to compute the inverse instructions, which are pushed into the process' heap. By adopting this two-passes approach, we have obtained several advantages, to abate time effort required by dynamic instrumentation and to overcome hurdles in solving complex control flow graphs to infer high-level software logic, which is needed to properly reverse instructions.

The proposal has been assessed on a real-world simulation model, targeted at the analysis of a GSM network of non-minimal size, in the presence of devices moving around the covered region. We have shown that, as the simulation complexity increases, the ratio between real simulation work process and reversible effort tends to a constant value. This indicates that the overhead grows much lesser than real work. As the second experiment demonstrates, reversible version allows to achieve very competitive speedups, though sub-optimal with regards to sparse state saving. Nevertheless, our approach guarantees a more stable gain for a wider range of simulation contexts, where the percentage of rollbacks increases or thwarts to find an optimal interval timing for checkpointing. Further, contrariwise to sparse state saving, our reversible rollback allows as well out of the box the correct execution of applications that are not Piece-Wise-Deterministic, still ensuring a considerable speedup.

Finally, we have investigated further improvements both in time and memory effort, and possible future applications fields, such as in debugging aiding tools. In fact, by the many similarities between state saving in the context of optimistic PDES, and reversible debugging, we have been able to devise a feasible work path to apply our current results to this little explored field.

Appendix A

The present section shows a working example of the ROOT-Sim framework usage in a context of a very simple simulation model. The example models a very mesh of nodes that randomly send events to each other. To correctly behave, it is mandatory to implement at least the two following callback functions: `ProcessEvent()` which undertakes incoming events, and `OnGVT()` which checks both whether termination condition is met and handle the commit frontier, it guarantees to eventually stop simulation.

Since the fairly simple model logic, the only two event type handled by `ProcessEvent()` are:

`INIT` It is the default event type to bootstrap the simulation process. When a `INIT` event arrives the framework allocates the LP's state.

`PACKET` It simply describes the delivery of a new message from some other process.

Any event type other than `INIT` is fully customizable by the user. In the present case, to the `PACKET` type is associated a simple routine that generates a new message to forward again. In this example some random distribution generator routines are employed, such as `Random()` and `Expent()`, which calculates the relative statistical distribution.

```
1  #include<ROOT-Sim.h>
2
3  #define PACKET  1 // Event definition
4  #define DELAY 120 // Application parameters
5  #define PACKETS 1000*1000 // Termination condition
6
7  typedef struct _event_content_t {
8    time_type sent_nt;
9  } event_t;
10
11 typedef struct _lp_state_t{
12   int packet_count;
13 } lp_state_t;
14
15 /**
16  * Callback invoked by the kernel whenever there is a scheduled
        and unprocessed event in the LP queue.
17  * Therefore the local process undertakes the event passed by
        performing user implemented logic.
18  * Note that the INIT integer event code is reserved to the
        library.
19  */
20 void ProcessEvent(unsigned int me, time_type now, unsigned int
       event, event_t *content, lp_state_t *state) {
21   event_t new_event;
22   time_type timestamp;
23
24   switch(event) {
25     case INIT:  // Reserved integer code to represent the
        initialization packet
26       state = (lp_state_t *) malloc(sizeof(lp_state_t));
27       state->packet_count = 0;
28       timestamp = (time_type) (20 * Random());
29       ScheduleNewEvent(me, timestamp, PACKET, NULL, 0);
30       break;
31
32     case PACKET:  // Custom integer code defining that a new
        packet is incoming
33       state->packet_count++;
34       new_event.sent_at = now;
35       int recv = FindReceiver(MESH);
```

```
36        timestamp = now + Expent(DELAY);
37        ScheduleNewEvent(recv, timestamp, PACKET, &new_event,
          sizeof(new_event));
38        break;
39    }
40 }
41
42 /**
43  * Callback invoked by the kernel whenever a frontier of commit is
        reached. Therefore the control is passed
44  * to the application layer. In this case the application will
        only checks if the termination condition is met.
45  */
46 bool OnGVT(int gid, lp_state_t *snapshot){
47    if(snapshot->packet_count < PACKETS)
48      return false;
49    return true;
50 }
```

LISTING 7.1: *Example for the ROOT-Sim environment*

Appendix B

Since integer division instruction are still costly for micro-code, in 1994 Granlund and Montgomery proposed an optimized alternative to perform integer signed and unsigned division without the straightway employment of the `IDIV` division machine instruction. They resorts to a sequence of addition, multiplication and shift instruction that realizes the same operational logic. The algorithm assumes a two's complementary machine architecture. In the case that multiplication instruction run faster than division, is convenient to realize integer divisions through multiplication by invariant. Table 7.1 offers an insight of the statistical employment of each arithmentic instruction in a generic program. It based from the static analysis Knuth performed on FORTRAN programs in one of his previous work [**?**].

There are other previous researches aimed to achieve faster integer division [**?**, **?**, **?**], though they only works for $2^k - 1$ dividend, and for relative small k values.

Nowadays most of the compilers adopts this optimization. GCC compiler collection has been updated from the .

Let us suppose to perform the following integer division in an N-bit two's complement architecture:

$$q = \lfloor \frac{n}{d} \rfloor \tag{7.1}$$

Where $0 < D < 2^N$, $0 \leq n < 2^N$ and N is the word size. The idea it to try to find a rational approximation $\frac{m}{2^{N+l}}$ for the divisor $\frac{1}{d}$, such that:

$$\lfloor \frac{n}{d} \rfloor \ = \ \lfloor \frac{m \cdot n}{2^{N+l}} \rfloor \qquad \forall\ 0 \leq n \leq 2^N - 1,\ 0 \leq l \leq N - 1 \tag{7.2}$$

Substituting in equation (7.2) the possible extents n could assume, d and the generic $q \cdot -1$, it follows that:

$$if n = d \rightarrow 2^{N+l} \leq m \cdot d if n = q \cdot d - 1 \rightarrow (m \cdot d - 2^{N+l}) \cdot (q \cdot d - 1 < 2^{N+l}) \tag{7.3}$$

Therefore, the following theorem 2 holds:

Theorem 2. *Let us suppose m, d, l are non-negative integers, such that $d \neq 0$ and*

$$2^{N+l} \leq m \cdot d \leq 2^{N+l} + 2^l \tag{7.4}$$

Then follows that

$$\lfloor \frac{n}{d} \rfloor \ = \ \lfloor \frac{m \cdot n}{2^{N+l}} \rfloor \qquad \forall 0 \leq n \leq 2^N \tag{7.5}$$

Theorem 2 holds because the maximum relative error (1 part in 2^N) is too small to affect the quotient when $n < 2^N$.

Operation	Percentage
addition	39%
subtraction	22%
multiplication	27%
division	10%
exponential	2%
total	46466 ins

TABLE 7.1: *Statistical arithmetic operation composition*

Operation	Signed	Unsigned
addition	1–8	–
subtraction	1–8	–
multiplication	5–9	4–8
division	22–47	17–41
shift	1–2	–

TABLE 7.2: *Number of cycles needed to perform arithmetic operations*

Appendix C

This appendix is dedicated to ELF object file format description, in order to give the reader a more valuable comprehension of what did in the present thesis work.

Overview

ELF (Executable and Loadable Format) file format was first proposed in the System ABI IV. ELF files can be of three types:

Executable which holds code and data suitable for linking

Relocatable which holds code page-aligned executable segments

Shared object which are binary files to be dynamically loaded and embedded in other programs

ELF files are produced by compilers during the different stages of computation.

Generally, those objects are structured in sections, that are segments of various size containing precise kind of data aligned to the following list. Relocatable objects are generated by the compiler as the output of the first compiling stage as input for the linker. This, in turn, extracts from the relocatable the relevant information to compute relocation's displacement for each entry and finally builds an executable file.

Structure

Figure 7.1 shows the structure of an ELF file in both variants, relocatable and executable. Accordingly to the purpose, it provides different information structured in a slightly different way.

ELF format is an offset-based file that embodies a compact and efficient representation for compiled binary programs.

Independently of the purpose, both files must have a ELF header section which tells how to access the remainder of the file. It holds information on registered sections or segments and where the relative header table is located, further is maintain low-level information on machine

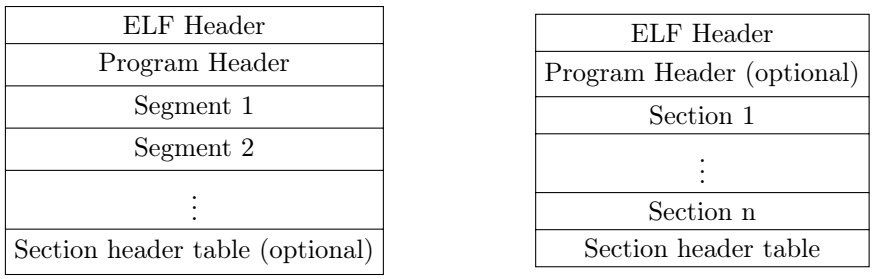


FIGURE 7.1: *ELF structure*

architecture. The program header is mandatory only for *executable* files and contains useful information on how to create the program's map to load in memory. Analogously in the sections header table are stored information regarding specifically registered sections, such as the size or the content type. Sections are the basic storage unit in ELF files. They contains several kind of data according to the specific purpose. A complete list of section's type are provided in Table 7.6. For the sake of brevity and according to the work's perspective, this appendix provides implementation details' description of the relocatable type only.

Table 7.3 describes data type wrapper used in the ELF structure. Types' name differ from 32-bit to 64-bit architecture for the 2 digits suffix; in the table, for brevity, are reported both architectures in the form of *Elfxx*, where "*xx*" represents either *32* of *64*.

Name	Purpose	32-bit Size	64-bit Size
`Elfxx_Addr`	Unsigned address	4	8
`Elfxx_Half`	Unsigned short integer	2	2
`Elfxx_Off`	Unsigned file offset	4	8
`Elfxx_Sword`	Signed large integer	4	8
`Elfxx_Word`	Unsigned large integer	4	8

TABLE 7.3: *32/64-Bits Data types*

Sections

Sections are the basic storage unit which contains code, string, data, references or other information needed to program to be linked and/or executed. Sections are registered in the *section header table* which holds one entry for each section. Section header's entries have all th same size, therefore it is straightforward to access them. Table 7.5 describes fields of the `Elfxx_Shdr` descriptor. Note that sections themselves are bulk container described by relative entries held by the `Elfxx_Shdr` table.

In the Table 7.9 are reported the most relevant special sections used in the present work.

String table

String tables simply hold an array of raw chars. To access them the it is sufficient to index the initial character to which to start. All the descriptor that involve a *name* field relies on this char-offset method to associate one entity to a string. This section starts and ends with a null character. Likewise, each string in the table end with a null character, in order to allow a proper parsing of the string itself.

Symbol table

Symbols entities are employed to correctly relocate symbolic reference within the code, such as instruction that affects memory locations, call or jump instructions, etc. Each symbolic reference in the high-level programming language is translated into a symbol entry, therefore compiler and linker can properly identify each entity. Thereby, each symbol in the ELF file is defined in relation to some other section to which it refers.

Field	Type	Description
e_ident	unsigned char [16]	Magic signature that identify an ELF file
e_type	Elfxx_Half	Identifies the file type, e.g. loadable, relocatable, shared object, etc.
e_machine	Elfxx_Half	Specifies the machine architecture
e_version	Elfxx_Word	Identifies the object version
e_entry	Elf_Addr	Hold the virtual entry point for an executable file, zero otherwise
e_phoff	Elfxx_Off	Specifies the offset from the beginning of file at which the program header table is located
e_shoff	Elfxx_Off	Specifies the offset from the beginning of file at which the section header table is located
e_flags	Elfxx_Word	Holds processor specific flags
e_ehsize	Elfxx_Half	Holds the section header's size in bytes
e_phentsize	Elfxx_Half	Holds the entry's size for the program header table
e_phnum	Elfxx_Half	Specifies the number of entries in the program header table
e_shentsize	Elfxx_Half	Holds the section header's size in bytes
e_shnum	Elfxx_Half	Specify the number of the entries in the section header table
e_shstrndx	Elfxx_Half	Holds the index of the entry in the section header table associated with the name string table

TABLE 7.4: *Header descriptor's fields*

Field	Type	Description
sh_nme	Elfxx_Word	It is section's name, specified as the string table entry's index within the section header
sh_type	Elfxx_Word	Describes section's type: see Table 7.6 for the main types
sh_flags	Elfxx_Word	Represents miscellaneous attributes according to the section's type
sh_addr	Elf_Addr	For executable files, this fields holds the virtual address at which the section resides, zero otherwise
sh_offset	Elfxx_Off	Holds the offset in bytes from the beginning of file where sections resides
sh_size	Elfxx_Word	Holds the section's size in bytes
sh_link	Elfxx_Word	According to section's type it assumes a different semantic, see Table 7.8
sh_info	Elfxx_Word	According to section's type it assumes a different semantic, see Table 7.8
sh_addralign	Elfxx_Word	Holds the address alignment
sh_entsize	Elfxx_Word	Specifies the size of each sections header's entry

TABLE 7.5: *Section descriptor's fields*

Name	Description
SHT_NULL	Inactive section
SHT_PROGBITS	Holds program information
SHT_SYMTAB, SHT_DYNSYM	Symbol table
SHT_STRTAB	String table
SHT_RELA	Relocation table. It supports Elfxx_Rela structure
SHT_HASH	Symbol hash table
SHT_DYNAMIC	Holds information for dynamic linking
SHT_NOBITS	Represents a SHT_PROGBITS with no size associated
SHT_REL	Old relocation table. It supports Elfxx_Rel structure

TABLE 7.6: *Section's types table*

Name	Description
SHF_WRITE	The section contains data written during execution
SHF_ALLOC	The section occupies memory during execution
SHF_EXECINSTR	The section contains executable code
SHF_MASKPROC	The section contains reserved for processor specific semantic

TABLE 7.7: *Section's attributes table*

Section's type	sh_link	sh_ info
`SHT_REL, SHT_RELA`	Index of the associated symbol table	Index of the section to which relocation applies
`SHT_SYMTAB, SHT_DYNSYM`	Index of the associated symbol table	Number of the local symbols

TABLE 7.8: *Section's `sh_link` and `sh_info` fields*

Name	Type	Attributes	Description
`.text`	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	Sections containing the executable binary code
`.data`	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE	This sections contains initialize data (e.g. global variables) involved in the program's map creation
`.rodata`	SHT_PROGBITS	SHF_ALLOC	Holds read-only data, such as embedded strings, or constants
`.rel`	SHT_REL	May include SHF_ALLOC if file is loadable	Contains relocation entries in the form of `Elfxx_Rel` structure entities
`.rela`	SHT_RELA		Contains relocation entries in the form of `Elfxx_Rela` structure entities
`.symtab`	SHT_SYMTAB	May include SHF_ALLOC if file is loadable	The section contains the symbol table
`.strtab`	SHT_STRTAB	May include SHF_ALLOC if file is loadable	The section contains raw strings as an array of characters
`.bss`	SHF_NOBITS	SHF_ALLOC + SHF_WRITE	This sections contains uninitialized data involved in the process' image creation

TABLE 7.9: *ELF's special sections description*

Name	Type	Description
`st_name`	Elfxx_Word	Holds the index corresponding to the first byte in the string table
`st_value`	Elf_Addr	In relocatable files this member holds the offset from section's beginning identified by `st_shndx` or the alignment constraint if this holds `SHN_COMMON`, namely generic section.
`st_size`	Elfxx_Word	Holds the symbol's size in the program semantic
`st_info`	unsigned char	Specifies both the symbol type and the binding attributes (which can be *local* or *global*)
`st_other`	unsigned char	Currently unused, holds 0
`st_shndx`	Elfxx_Half	Index of the section at which the symbol belongs

TABLE 7.10: *Symbol descriptor's fields*

Name	Description
`STT_NOTYPE`	The symbol type is not otherwise specified
`STT_OBJECT`	The symbol is associated with an object (variable, array, etc.)
`STT_FUNC`	The symbol refers to a function or executable code
`STT_SECTION`	The symbol refers to a section within the ELF file

TABLE 7.11: *Symbol's type table*

Relocation

Relocation connects symbolic references to relative definitions. For example a call instruction to a specific code region is embedded through a relocation entry which bind the specific instruction towards its destination by means of a symbolic reference. There are two structure: an older `Elfxx_Rel` and `Elfxx_Rela`. This section provide a table fields descriptor for the latter only.

Field	Type	Description
`r_offset`	`Elfxx_Off`	Holds the location to which the relocation applies, either as an offset from the beginning of the section affected, or as a virtual address
`r_info`	`Elfxx_Word`	Specifies both the the symbol table index with respect to which relocation applies and the relocation's type
`r_addend`	`Elfxx_Sword`	Holds a signed offset value used to compute final relocation value

Table 7.12: *Relocation descriptor's fileds*

Appendix D

Herein is reported the complete assembly code snippet relative to Chapter 4 presented in Section 4.1 "The reversing code approach". For the sake of completeness here is likewise reported the C code as well. Listing 7.3 illustrates the assembly output of the GCC-4.9.2 compiler. We have added inline to the assembly code also the C statements to allow a fast binding between the logic operation and the relative assembly idiom.

As one can easily observe, GCC compiler dose several non-trivial optimizations that render code quite weird to understand. Moreover integer division does not employ any division assembly instruction, instead a multiplication is performed. The reason behind this kind of optimization is discussed in Chapter 7 "Appendix B".

C source code is listed below:

```
1  int foo(int num) {
2    int x;
3
4    x = 10;
5    x *= 2;
6    x++;
7    x = x >> 2;
8    x /= 3;
9
10   x--;
11   x *= 3;
12   x = num + 8;
13   x %= 2;
14
15   return x;
16 }
17
18 int main(){
19  foo(5);
20  return 0;
21 }
```

LISTING 7.2: *C source code of a generic function*

In the following the relative assembly translation is provided:

```
1  0000000000000000 <foo>:
2  int foo(int num) {
3     0:   push   %rbp
4     1:   mov    %rsp,%rbp
5     4:    mov    %edi,-0x14(%rbp)
6          int x;
7
8          x = 10;
9     7:   movl   $0xa,-0x4(%rbp)
10
11         x *= 2;
12    e:   shll   -0x4(%rbp)
13
14         x++;
15   11:   addl   $0x1,-0x4(%rbp)
16
17         x = x >> 2;
18   15:   sarl   $0x2,-0x4(%rbp)
19
20         x /= 3;
21   19:   mov    -0x4(%rbp),%ecx
22   1c:   mov    $0x55555556,%edx
23   21:   mov    %ecx,%eax
24   23:   imul   %edx
25   25:   mov    %ecx,%eax
26   27:   sar    $0x1f,%eax
27   2a:   sub    %eax,%edx
28   2c:   mov    %edx,%eax
29   2e:   mov    %eax,-0x4(%rbp)
30
31         x--;
32   31:   subl   $0x1,-0x4(%rbp)
33
34         x *= 3;
35   35:   mov    -0x4(%rbp),%edx
36   38:   mov    %edx,%eax
37   3a:   add    %eax,%eax
38   3c:   add    %edx,%eax
39   3e:   mov    %eax,-0x4(%rbp)
40
```

```
41          x = num + 8;
42    41:   mov    -0x14(%rbp),%eax
43    44:   add    $0x8,%eax
44    47:   mov    %eax,-0x4(%rbp)
45
46          x %= 2;
47    4a:   mov    -0x4(%rbp),%eax
48    4d:   cltd
49    4e:   shr    $0x1f,%edx
50    51:   add    %edx,%eax
51    53:   and    $0x1,%eax
52    56:   sub    %edx,%eax
53    58:   mov    %eax,-0x4(%rbp)
54
55          return x;
56    5b:   mov    -0x4(%rbp),%eax
57  }
58    5e:   pop    %rbp
59    5f:   retq
60
```

```
61  0000000000000060 <main>:
62  int main(){
63    60:   push   %rbp
64    61:   mov    %rsp,%rbp
65
66          foo(5);
67    64:   mov    $0x5,%edi
68    69:   callq  6e <main+0xe>
69
70          return 0;
71    6e:   mov    $0x0,%eax
72  }
73    73:   pop    %rbp
74    74:   retq
```

LISTING 7.3: *Assembly translation of GCC-4.9.2 compiler*