SAPIENZA
UNIVERSITÀ DI ROMA

Master Thesis in

ENGINEERING IN COMPUTER SCIENCE

Lightweight approximate virtual page access tracing of multi-threaded applications via static binary instrumentation

Advisor

Prof. Francesco Quaglia

Candidate

Simone Economo

Co-Advisor

PhD. Alessandro Pellegrini

Academic Year 2014/2015

Acknowledgment

A thesis, be it a bachelor's thesis, a master's thesis or any other form of dissertation, is not a thesis without acknowledgments. First of all, because we should never lose the occasion to thank those we love and esteem. Second, because many theses wouldn't be possible without the professional and emotional support of many different people—this couldn't be more true for my thesis. Last but not least, because they are *so damn fun* to read. So, here we are!—beware: this chapter makes ridiculous use of *italics*, don't say I didn't warn you!

My sincere gratitude goes above all to Prof. Francesco Quaglia and Prof. Bruno Ciciani for introducing me to the amazing world of computer architectures and operating systems, and for turning me into a better *engineer* through their wonderful courses. They are undoubtedly inspiring from a professional point of view, but I mostly *admire* them as persons. I also wish to thank PhD Alessandro Pellegrini for his support, especially in the last *crazy* days before the graduation date, and for all the incredibly *nerdy* chats we had between one "segmentation fault" and another.

I owe a big "thank you" to Davide too, a kind friend and colleague. Your help has been *invaluable* to me and I am extremely grateful to you for sacrificing *your* time in my *darkest* hours. I wouldn't be here if it wasn't for you.

I cannot be more grateful to Lorenzo, Marco and Elisa for being the best friends one can ever have. One way or another, you made me feel lucky through your words, thoughts and actions—Marco's *langiullate* deserve honorable mention... they're memorable! I am *proud* to have you in my life. Thanks also to Donato, Lorenzo N., Martina and Federica for being a late but extremely *welcome* addition to this list.

To my friends of the exclusive "Frati Francescani del DIS" club, which also happen to have founded the *coolest* Whatsapp group ever, "Le Panze Sacre", I wish to say: do you already have a schedule ready for next's month list of *sagre* and food festivals?! Jokes apart, thanks for all the amazing moments that we had and, hopefully, will have in future. No, seriously... *sagra del tartufo* anyone?

As for the emotional support, I cannot help but express my deepest gratitude to the people that hold a special spot in my life. My parents, Giancarlo and Ileana, because they have always seconded my decisions and supported my choices in every situation, even when they were a bit... doubtful. I hope to be able to return to you at least a tiny portion of the *boundless* love that you constantly give to me. My relatives, too, because they made me feel part of a bigger family. Last but not least my second family, Annalisa, Fabrizio and Diana, for teaching me the real meaning of *happiness*—but also for pleasuring my stomach with delicious food!

Have I forgotten someone? I don't think so... that's all folks, you can stop reading now!

To Claudia, I wish to say more things than I will ever be able to write. We joked on the number of lines that you *deserved* in the acknowledgments, like if you deserved a number! What you did for me, in these months... since we met, it is something that make me feel loved in a way that is special and unique. I see you as my half and my dearest friend. Someone to protect, but also my strength. You complete me, but also challenge me in all positive ways. You are the proof that a simple heart can be full of unimaginable love, and I'm glad that, six years ago in *Aula 16*, you decided, completely out of *your* mind, to take *my* hand for the first time. *Thank you.*

To Alessandra and Orietta,
for all the love that you gave to me.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

If the auto industry advanced as rapidly as the semiconductor industry,
a Rolls Royce would get half a million miles per gallon,
and it would be cheaper to throw it away than to park it.
— Gordon Moore (1998, attributed)

Computing architectures are complex. In the early days of computing, systems were simple. When it was clear that the original design could not provide anymore the performance guarantees and the functional capabilities demanded by industry, such design underwent a change. On the one hand, the need for accelerated computer graphics resulted into dedicated graphic cards, bundling special-purpose cores with support for unconventional operations that can offload most of the floating-point-intensive tasks from the main processor. On the other hand, the demand for powerful portable devices with long-lasting batteries caused small power-efficient cores to be placed side by side on the same chip with large high-performance ones. Nowadays we have computing systems which are powerful and highly performing, but also able to consume as little energy as needed to meet economical constraints. Moreover, they are heterogeneous, meaning that their ability to work at different performance and power level requirements is provided by a wide spectrum of complementary components, each possessing different operational capabilities and peculiarities.

The road to *heterogeneous systems* was driven not only by industry pressure, but also by physical constraints, i.e. 'walls' that deflected the expected course of evolution of computing architectures in an unforeseeable, yet fascinating way. The *power wall*, historically speaking, is the first of such walls. It states that individual computing units cannot be made faster without violating the limits imposed by fundamental physics via the power equation [1]:

$$P_{DD} = \alpha \cdot C V_{DD}^2 f_{av} \tag{1.1}$$

In the above equation, P_{DD} represents the *average dynamic power* dis-

sipated by a single CMOS transistor in response to an input change. When the input value switches from high to low or vice versa, the internal capacitance needed to keep the circuit output stable gets charged or discharged. For this reason, the above quantity depends on the capacitance load C, the supply voltage that drives the circuit V_{DD} and the average frequency of input events f_{av}. Additionally, an activity factor α is included to represent the probability that the circuit's internal capacitance state will switch upon an input event.

Clearly, the higher the power dissipated by a transistor, the higher the heat which is eventually radiated as a by-product, which in turns requires more powerful (and expensive) cooling techniques. As we can see from equation 1.1, the only way to decrease the dynamic power P_{DD} consumed by a circuit is by decreasing the value of all other quantities involved. However, a decrease in the value of f_{av} means a slower clock rate which, in turns, yields a slower system. Therefore, to benefit from higher clock rates without affecting the dissipated power, C and V_{DD} must decrease.

If we look at Moore's Law [2], we observe that from 1970 to 2004 the number of transistors in a computing system doubled roughly every 18 to 24 months, while the clock rate increased every 34 months (see figure 1.1). These trends can be explained by virtue of the power equation, since a decrease in transistors size means a decrease in load capacitance. Hence, by scaling down the size of transistors, processor manufacturers were able to provide more horsepower at the same or slightly increase consumption of power and heat radiation. At the same time, by reducing the supply voltage of circuits, faster clock rates were possible. Over time, Moore's Law turned into a self-fulfilling prophecy, driven by hardened competition and the demand for faster processors.

Unfortunately, circuit sizes and supply voltage cannot be driven arbitrarily down. The leakage current which is responsive for the static power consumption component of circuits becomes more and more influential as transistors shrink. Moreover, circuits become more and more sensible to interfering signals as supply voltage approaches the same order of magnitude as noise. Eventually, one cannot make transistors smaller or supply voltage lower, while also increasing the clock rate and keeping dynamic power consumption below acceptable thresholds. One of them must be sacrificed.

This led to the concept of *multi-core* systems, i.e. computing architectures where multiple processing units are present and operate in a parallel fashion, at a lower individual clock speed than single-core systems but providing overall an increased computing power. By giving up individual clock speed, the process of scaling down the size of transistors can continue. Besides scaling in the vertical direction, increasing the number of processing units allows to increase the overall number of transistors by virtue of hori-

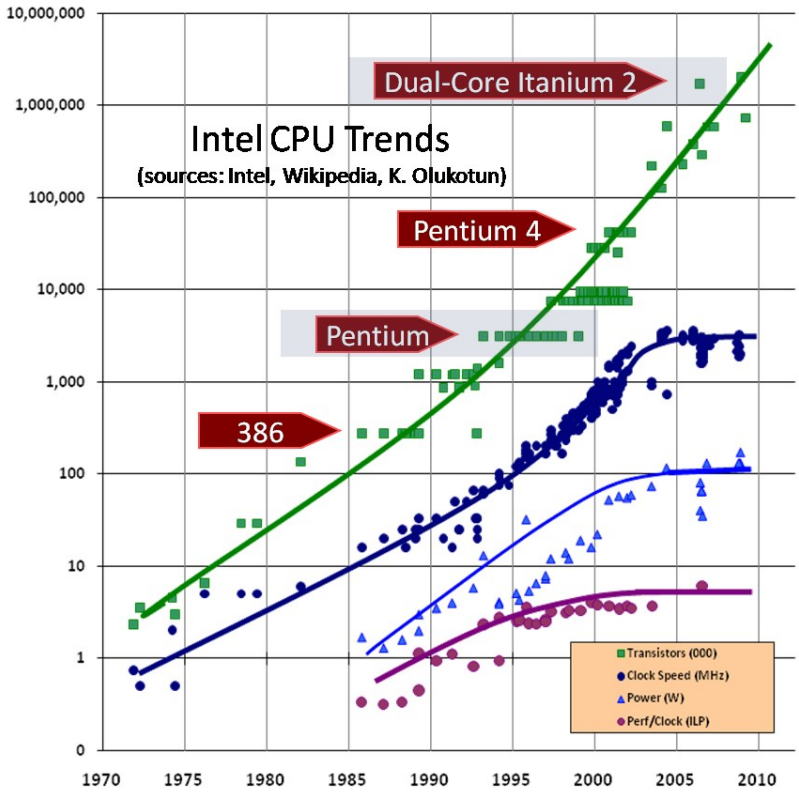

FIGURE 1.1: *A pictorial representation of Moore's Law from 1970 to 2010.*

zontal scaling, thus keeping up with Moore's Law.

The *memory wall* is another of such figurative walls. Due to technological constraints, processor speed began to increase at a faster rate than memory bandwidth. This ever-increasing imbalance turned out to drastically hamper the performance of even highly-optimized scientific applications [3]. As a result of this trend, processors started to bundle smaller and faster memories closer to their cores, the so-called *caches*. The introduction of cache memories was an important breakthrough for computing architectures, since for the first time it invalidated the statement that all memory accesses are equal. Nowadays, the concept of memory caches is so pervasive that additional memory layers were gradually added, resulting into what is referred to as the *memory hierarchy*.

Similarly to the power wall, the *coherency wall* has been hit not so ago. It results from increasing the number of cores and the raw storage capacity in a system up to a point where: (a) it becomes economically prohibitive to maintain an unified shared-memory model with growing storage; (b) cache-level synchronization protocols are the main source of bottleneck for programs.

On the one hand, large-capacity memory banks are still too expensive to produce with the current manufacturing technology. They are easily defeated by summing up the capacities of smaller and cheaper memory banks. By grouping cores in clusters and spreading such memory banks in such a way that each cluster has its own memory banks, memory access latency for local requests can be kept low to the detriment of remote accesses which are more costly. Moreover, this allows the system to spread accesses across different physical storage components, thus reducing contention at memory buses. This is the model adopted by NUMA architectures (Non-Uniform Memory Access) [4].

On the other hand, the presence of a high number of cores, hence of many disjoint cache memories, poses challenges to any synchronization scheme aimed at enforcing data consistency. Many-core systems with a single memory bus usually implement a *bus snooping protocol*. According to this scheme, every write request on a private cache memory produces a broadcast event on the bus which forces all other caches targeted by the update to invalidate their lines. For read requests that produce a cache miss, a request message is also broadcast. This kind of architectural scheme would not work in a many-core scenario because of the high probability of contention at the interconnection level, even with the presence of multiple memory banks and buses.

For this reason, many-core platforms implement the *directory-based protocol*. Whenever a core loads a cache line in its private cache, it registers itself into a shared global directory. In this directory, for every cache line, there's a list of registered cores that expect to be notified upon a change on that line, by means of an invalidation message, or a read request. This mechanism scales better than the snooping bus, but it can still create contention at the various interconnection buses if the number of cores that share a line increase. For this reason, cache-incoherent architectures are being studied where coherency of cached data along different clusters is not automatically enforced by the firmware [5].

This thesis studies an ancient but still relevant problem in computer science. Namely, the detection of the memory access pattern exhibited by generic applications—both single-threaded and multi-threaded ones. Memory access traces have several applications both for performance and profiling purposes. As an example, traces can be used together with a simulation environment to assess the degree to which the application is able to efficiently interact with the underlying memory hierarchy of the architecture, as complex as it can be. The same kind of traces can also be used to guide hardware design, in the long run, so as to more closely match the behavior of a desired class of scientific applications.

Actually, the work presented in this thesis is a specialization of tracing

to virtual pages. To the best of my knowledge, virtual page access tracing is a novel problem in the literature which, like generic memory tracing, has many interesting applications. In the heterogeneous systems scenario, tracing of virtual pages can be used to dynamically move data in a NUMA architecture closer to the cluster where the thread is actually running, or moving the thread closer to the NUMA node where its working set is actually residing [6] [7]. To this extent, the target application can be both a generic multi-threaded program or an application that runs on top of a custom middleware, such as an optimistic simulation platform [8]. In this case, moving data closer to a thread means tracing the state of a simulation object not only to migrate it, but also logging it into a close memory bank to support efficient rollback mechanisms [9].

The key technique to accomplish virtual page access tracing is *binary instrumentation*, which is a way to transparently execute additional logic at run-time without affecting the actual application-level behavior coded by the programmer. Historically, instrumentation has been seen as a tool for debugging and profiling purposes. Nonetheless, general-purpose instrumentation is starting to emerge in the literature as the means to implement any sort of additional logic, apart from that bound to maintenance and monitoring tasks. Notice that instrumentation tends to work at the application-level, thus leaving unaltered the library and operating system levels. This allows instrumentation tools to work 'out of the box', without any modification to the compilation tool-chain, the standard library environment or the operating system. However, it is natural to look at the problem the other way round—namely, from the point of view of the standard library and the OS. Even as instrumentation requires no adjustment at these levels and other consolidated development tools, these layers and tools alone can provide virtually the same services as instrumentation. Furthermore, they can do it with the same or even higher degree of transparency of instrumentation—since no additional compilation pass to the application must be made,—provided that we're prone to developing patches for them. The naturally rising dilemma is hence to choose one approach or the other, aware of the benefits and drawbacks that come along with this choice.

To this extent, my work makes extensive use of instrumentation to the detriment of the complementary methodology based on internal standard library, kernel and compiler changes. The positive effect of the instrumentation approach, as will be clear later in this thesis, is that it allows for great precision at the cost of a slightly increased run-time overhead for acquiring traces. This shouldn't be much surprising, given that we are perturbing the baseline performance of the application with additional instructions that serve an analysis purpose, and not the application's business logic goal. Reducing this instrumentation overhead, while affecting the accuracy to the

minimum extent, is a central topic in this thesis which will also be part of more deeper investigations in the future. Another relevant topic is that of controlling the balance between instrumentation accuracy and instrumentation overhead so that different analysis need can be meet. This, however, depends on the particular program being instrumented, the current behavior of the program at run-time and possibly the presence of particular external conditions. Once again, the importance of this topic is recognized in this thesis, but I've only started to to scratch its surface.

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of instrumentation techniques and existing tools. It will then focus on static binary instrumentation and Hijacker, a C toolkit designed to achieve general-purpose static instrumentation through lightweight and non-intrusive injection techniques. Chapter 3 builds on the concepts presented in the previous chapter to give a general perspective on the memory access tracing problem, as well as the most important techniques employed in the literature. Chapter 4 is the heart of this thesis and explains how one can solve the problem of tracing virtual pages by leveraging on finer-grained memory tracing techniques. Chapter 5 provides the reader with an experimental and quantitative justification of the goodness of my approach in terms of instrumentation overhead and tracing accuracy. Finally, chapter 6 lays out the conclusions of this thesis and anticipates possible directions for future work.

CHAPTER 2

Instrumentation

Your eyes can deceive you. Don't trust them.
— Obi-Wan Kenobi, A New Hope (1977)

Instrumentation is the art and science of measurement and control of the behavior of a program, achieved in such a way as to be transparent to the functioning of the program itself. At its bare minimum, it entails augmenting an application with additional logic and data, while leaving its semantics untouched. Upon running, the program looks and behaves as if it no changes were made to it. Underneath, the actual program behavior has been extended with background functionalities aimed at a variety of different goals. The traditional application of instrumentation techniques is for profiling and monitoring purposes, but other usages are possible. Some interesting uses of instrumentation are to check for programming errors in the code, as well as security breaches. It can also be employed to review the performance of an application, optimize it for future runs and even guide the design of hardware and system components. Generally speaking, instrumentation is a technique that is frequently used to achieved the following goals:

- Profiling the evolution of a program over time in terms of memory access pattern, interaction with the cache subsystem, libraries and the operating system, mostly for optimization purposes, performance modeling and performance prediction.
- Debugging the execution a program to detect unrecoverable errors (e.g. division by zero), security vulnerabilities (e.g. buffer overflows) and correctness violations (e.g. in terms of program state predicates).
- Logging of higher-order application-specific metrics, such as committed vs. aborted transactions in a transactional system, or the number of invalidated events in a discrete event simulation (DES) platform.

- Virtualization and sandboxing, as a way to protect the underlying operating system and run-time environment from malicious code, as well as to decouple the request for software and hardware resources from their actual exploitation.

Instrumentation approaches can be classified at first according to the adoption of either hardware or software tools. As for software instrumentation tools, a natural way to classify instrumentation techniques is: (a) by the software stack level at which they operate—i.e., operating system, environment or application,—(b) by the actual level of abstraction seen by these tools—i.e., source code vs. machine code—and, (c) by the use of either static or dynamic approaches.

2.1 Hardware instrumentation

Hardware instrumentation is characterized at the architectural level by the presence of additional hardware components which are able to perform certain low-level profiling tasks with almost zero-overhead with respect to software-based equivalents [10]. The most common form of hardware instrumentation is high-resolution hot spot analysis and was introduced by the z10 family of IBM processors along with the z/OS proprietary kernel. Finding hot spots in the code allows to understand which portions of the code are run more frequently, so that they can be further optimized for performance purposes.

Interestingly, hardware instrumentation is fully transparent both to the application developer and to the application code. Since all the instrumentation logic occurs in firmware which moves instrumentation data to special-purpose hardware registers, the application code needs no changes. This kind of instrumentation, albeit interesting, is quite rare outside of mainframes and embedded systems. As such, it won't be further investigated.

2.2 Software instrumentation

Software instrumentation approaches are by far the most common. Program patching is performed by injecting additional instructions and data into the original program (or by modifying existing ones) in such a way to implement the desired functionality. This form of patching partially retains the transparency property with respect to the application developer, but requires an additional instrumentation step either at compile-time or run-time (or both). Moreover, it may or may not be transparent to the

application code itself, depending on the specific kind of instrumentation technique being used (i.e., static or dynamic).

A nice property of software instrumentation is that it allows to instrument software located at different logical levels, ranging from the operating system to the final application, including middlewares and libraries. Operating system instrumentation is possible but is rarely employed, as it would require patching a kernel image that follows completely different rules than common executable, but also because it is much easier to extend the system with dynamically-loaded modules. It is by far more common to instrument either the support environment on top of which user-level programs are built, or directly the program themselves. Conceptually, the two choices can be rather similar, it depends on whether the programming language is actually accompanied by an external run-time (as for interpreted languages and just-in-time compiled ones) or relies on traditional compilation techniques. In the latter case, instrumenting the environment means patching one or more libraries which are explicitly invoked by the compiled program.

When talking about patched code, we must specify the level of abstraction spoken by the code itself. To this extent, software instrumentation can occur at three different levels: machine code level, byte code level and source code level.

Machine code level At machine code level, programs are represented by a sequence of instructions and data bytes with no semantics. Instructions are strongly typed in terms of the number of bytes that are involved in an operation. At the same time, data items are seen as a sequence of bytes with no meaning, along with a size. As a result, this layer provides no clues as to how a program is structured, as well as to how data objects in memory must be interpreted. Hence, it is not suitable for high-level instrumentation tasks such as those targeting high-level control flow structures (e.g. `try-catch` or `switch-case` statements) or high-level data types (e.g. `struct`'d or `union`'d types, classes, callbacks, etc.) On the other hand, carrying out monitoring tasks, such as detecting the memory access pattern of an application, is extremely easy at this level, since all memory operations are explicit. Moreover, this level allows to instrument pretty much every executable resulting from a program written in a compiled programming language, regardless of its syntax and constructs.

Byte code level At byte code level, programs are encoded into an intermediate binary representation which includes a minimal amount of semantic information concerning control flow structures and complex types. This level is provided by interpreters and just-in-time (JIT) compilers for highly abstract languages that aren't directly compiled into machine code. As such,

it is only available in restricted contexts and depending on the programming language in which the application logic is described. The only additional benefit, with respect to the machine code level, is that instrumentation tasks at this level are independent from the architectural idiosyncrasies of the machine. As such, they are transparently compatible with whatever Assembly encoding for instructions is adopted by the hardware.

Source code level At source code level, programs are fully described by extremely complex structures and detailed meta-data. The level of precision that can be achieved at this level in terms of program semantics is the highest of all other levels, and portability of instrumentation tasks is at its finest. On the other hand, injecting instrumentation code within the application is much more complex, since the instrumentation logic must deal complex program statements and data types. At the same time, this kind of instrumentation requires the availability of the original source code file, whose absence makes it impossible to carry out the desired task.

As already anticipated, instrumentation can be a compile-time issue or a run-time one. Techniques that fall to one side or another are respectively called static techniques and dynamic ones.

2.2.1 Static instrumentation

In *static instrumentation*, changes are applied at the end of the compilation process, thus affecting the execution of the program 'once and for all'. As a result, the instrumentation cost is spent only once, ahead of execution. Observe that static instrumentation is not transparent to the application, since instrumentation code and data are directly inserted into the final executable file being executed. Furthermore, it is not able to fully capture the non-deterministic nature of applications. The program's actual control flow cannot be seen and it can only be modeled through a partially-ordered set of states.

The two main static instrumentation techniques are referred to as relocation and inlining. In *relocation*, the code to be instrumented is duplicated and inserted into a new text segment where it can be freely instrumented. In order for the instrumented code to be run in place of the original one, the original code must be patched with new branching instruction which need to be put at appropriate points in the original code called *instrumentation points*. When an instrumentation point is reached at run-time, the branching instruction diverts the flow to the instrumented region. Then, as soon as the execution of instrumented code completes, another branch diverts the flow back to the original code, until another instrumentation point is met.

This technique is welcomed by researchers because the amount of changes that need to be applied to the original code are minimal. The addresses of non-instrumented code regions need not be changed, therefore references to locations in these regions remain intact. The only requirement is that the instruction which holds the instrumentation point prior to patching the original code be large enough as to accommodate a branch, without overwriting the bytes of a subsequent instruction. This is as simple as it sounds for RISC architecture, but becomes a critical problem in CISC systems. Moreover, the overhead induced by relocation can be measured in terms of two additional branches for each instrumentation point. This can quickly become an issue in modern architectures, since branching instructions can lead to pipeline stalls.

Inlining takes a quite different approach than relocation. Original code is never copied, but rather instrumented in-place by inserting instrumentation code at the desired points, while shifting all instructions that follow. As such, no original instruction in the code is ever replaced—unless the specific instrumentation task to be carried out explicitly asks so. The cost of branching instructions is completely avoided, thus producing a more lightweight instrumentation. On the other hand, all original references to code need to be updated, included those that are indirectly encoded in data sections—e.g., jump tables for `switch-case` statements and function pointers hold by variables. This task can be more difficult the less meta-data is available in the original executable file to detect or infer with enough confidence the presence of such indirect references.

2.2.2 Dynamic instrumentation

In *dynamic instrumentation*, also called *JIT instrumentation*, program execution is patched at run-time, on the basis of the occurrence of particular run-time conditions. This latter approach tends to place a bigger burden on the performance of the instrumented application, since the cost is spent in minimal quantity ahead of execution and heavily at run-time. Also, compared to the static approach, dynamic instrumentation tends to be more opaque to the application developer, since dynamic instrumentation tools usually come in the form of virtual machines (VM) and emulators, that must be explicitly invoked by the user upon launching the client code. However, dynamic instrumentation achieves higher precision than static instrumentation when it comes to tracking the evolution of a program over time, since the control flow is observed on the fly and becomes a streamlined sequence of encountered program states.

The vast majority of dynamic instrumentation toolkits instruments code by either running it natively on the architecture or by means of emulation.

In *native execution*, chunks of the original application are executed on the underlying architecture one at a time. Upon being launched, the instrumentation virtual machine (VM) loads the passed client program into its same address space, intercepts the first instruction of the program and pulls out the first chuck of code, rooted at that instruction. The last instruction of this newly-created chunk is patched so that after the actual execution of that chunk, control is passed back to the VM. The next chunk to be executed is then created, starting from the next instruction which was scheduled for execution prior to patching. During the creation of a chunk, code is instrumented according to the requested rules and external processing functions are possibly invoked for the sake of analysis.

To speed-up execution, a software code cache is maintained where already created chunks are stored. Whenever the VM re-gains control over the application, it first checks whether the next scheduled chunk already exists by looking up in the code cache. In the positive case, the chunk is retrieved and the client program resumes. Another useful technique, named *linking*, allows to completely bypass the cache look-up mechanism while the current chunk is being created by the VM whenever the next chunk to be executed is already known to be in the cache. In such a case, it is not necessary to invoke the VM dispatcher because the VM itself already knows that the look-up will succeed. For this reason, the last instruction of the current chunk and the first instruction of the next chunk are directly linked.

To ease the task of instrumenting applications in a machine-independent way, some JIT VMs create an *intermediate representation* (IR) of the code in each chunk where architectural-dependent details are abstracted away in favor of a more explicit and general low-level language (similar in purposes to byte code). This intermediate language features an arbitrary number of virtual registers and explicitly represents the side-effect produced by original machine-code instructions. On the one hand, virtual registers allow to conceal certain register movement operations caused by the high register pressure of the underlying architecture. On the other hand, annotating the side-effect of each instruction makes it possible to the VM to keep track of all the invisible effects that come from its execution. Instrumentation that is performed on the IR is architectural-agnostic and gets implicitly converted into actual machine code at a later stage, when machine-specific optimizations are also performed. Observe that the increased expressiveness inherited from the IR is payed in terms of instrumentation overhead, since the first time a chunk is met the IR must be created, too.

The major drawback of the native execution technique is that it needs to intercept all the instruction that diverts the natural execution flow of the application. These instructions include direct and indirect branches, but even instructions that call privileged services—e.g., through software

interrupts—or generate exceptions. Not only that, but other library and kernel-level 'curlicues' must too be handled. This include asynchronous invocation of callbacks—like the thread-level messaging system of Windows or POSIX's signal handlers—as well as other subtle schemes—such as Pthread's thread creation or the `setjump`/`longjmp` library services.

Emulate execution is a technique which is mostly adopted in contexts where security is paramount and client code cannot be trusted. The major difference with respect to the native counterpart is that every machine-level instruction is interpreted, and a virtual processor state is manipulated by the VM which never relinquishes control. The original application therefore is never really executed, but can still be instrumented while its execution is being emulated by the VM. This approach clearly produces an overhead which is degrees of magnitude higher that the other form of execution. Moreover, it becomes quite complex to implement when the ISA that must be emulated is complex too—e.g., x86 and x86-64. Nevertheless, it is more secure than native execution and can provide a sandboxed environment in which the execution of the client program is evaluated without affecting other part of the system.

In my thesis, I will stick to software binary instrumentation techniques, since the kind of task which I wish to carry out—basically, tracing a subset memory accesses—is a natural low-level functionality. As such, it is efficiently performed at machine code level. Moreover, since I want this task to be completed in the most lightweight and less intrusive way—so as not to hamper the performance of the application—static techniques are preferred to dynamic ones.

2.3 Hijacker

Hijacker is the static binary instrumentation tool of choice for my thesis. It has been developed at the *Department of Computer, Control and Management Engineering* "Antonio Ruberti" at *Sapienza University of Rome* for high performance computing (HPC) settings [11]. Its main features are the ability to support *general-purpose instrumentation*—i.e., not bound necessarily to profiling or monitoring purposes—and the fact that it poses minimal code overhead on the target applications—a property which is highly desirable in HPC scenarios.

Another peculiar feature of Hijacker is its exclusive support for relocatable files, as opposed to executable ones. The main difference between relocatable files and executable ones is that executable files can be directly loaded into a process' address space, while relocatable files are typically not amenable to loading and must pass through an additional linking step. More precisely, an executable file is the union of one or more relocatable files, as

performed by the static linker during the compilation process. Hence, the main advantage gained from instrumenting a relocatable file is that it allows an instrumentation tool not to take care of any complex linking logic. Once again, this is the approach taken by Hijacker. A pictorial representation of the instrumentation process performed by Hijacker on relocatable files is shown in figure 2.1.

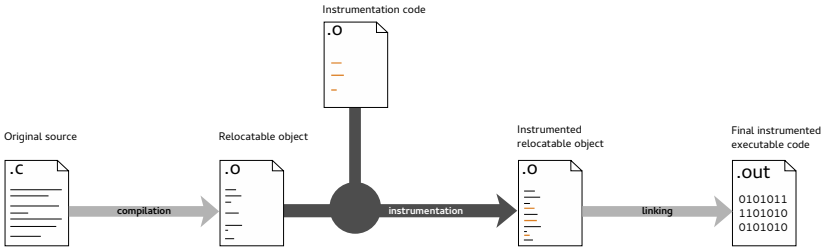

FIGURE 2.1: *Hijacker's instrumentation process*

In relocatable object files, the program is seen as a sequence of *sections*, each of which contains a different kind of information. Some sections contain program code, some program data. Other sections are only needed to guide the linking process and won't make through this step. All references to objects and places within each section are expressed as displacements from the beginning of the section and sometimes accompanied by detailed annotations called *relocations*. Therefore, relocatable objects never refer to functions or variables through virtual addresses, but rather through displacements and relocations. When the linked is fed with a relocatable object, it consumes all relocations and converts the relative addresses represented by displacements into absolute addresses, which are directly mapped into the virtual memory as virtual addresses. Those sections that were only needed for linking are discarded and the remaining ones become *segments* which are amenable to being loaded into virtual memory at program launch.

As with other compiling and instrumentation tools, Hijacker relies on the notion of intermediate representation. In some compilers, such as the LLVM compiling suite [12], a program source code is translated into an intermediate low-level representation that flattens complex control and data structures into simpler equivalents, as machine code does, while retaining the property of being machine-independent. Because of this, every optimization and manipulation that can be performed on a client program is done at the intermediate level, and translation to the final machine code is only performed as the last compilation stage. In Hijacker, the intermediate representation has similar but not exactly equivalent purposes. In fact, it is a representation for relocatable files rather than for source code, meaning that it creates an abstract representation of relocatable files in terms of sections, relocations and other entities.

The internal software architecture of Hijacker is depicted in figure 2.2. From an extremely high-level perspective, Hijacker is the combination of a front-end module and a back-end one. The *front-end* is responsible for parsing an input relocatable file and emitting another relocatable file which results from instrumenting the former. The *back-end* module takes care of the actual instrumentation tasks and is responsible for maintaining and altering the intermediate binary representation (IBR).

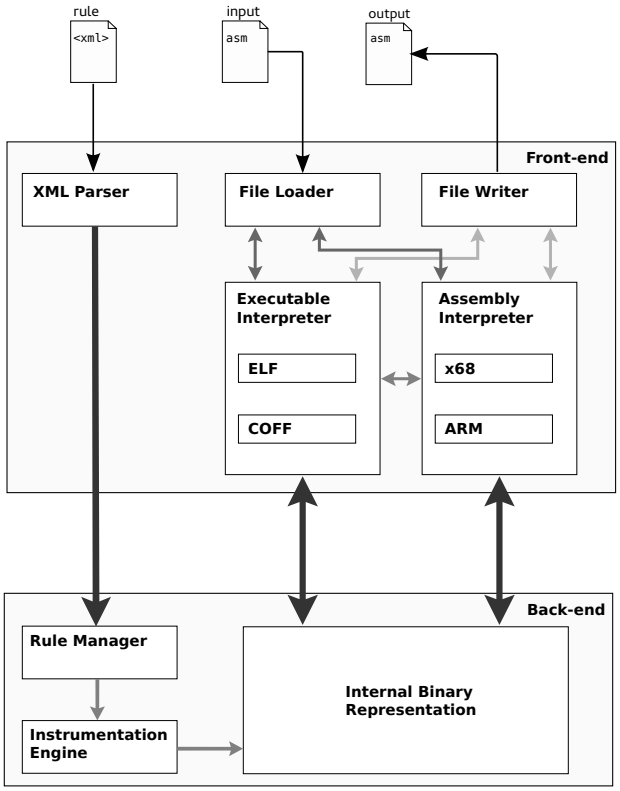

FIGURE 2.2: *Hijacker's software architecture*

2.3.1 Front-end

The front-end is the first and last module to take control of the entire instrumentation procedure. It is divided into three additional components:

- The *file loader* parses the input relocatable file and creates an internal mapping of sections, relocations and other entities (e.g. functions and instructions) where displacements from the beginning of sections are replaced by memory pointers. Eventually, the result of this operation is the IBR that will be the subject of all manipulation throughout the entire instrumentation process.

- The *rule parser* takes as input an XML file containing the requested instrumentation rules to apply to the relocatable file. It is specified by the user and its existence represents the support for general-purpose instrumentation tasks. Specifically, rules are registered and fed to an back-end subcomponent which is in charge of executing them in sequence.

- The *file writer* has the opposite role of the first. It converts the modified IBR into the same binary encoding that were used to describe the original relocatable file. As such, memory pointers are converted again into (updated) displacements from the beginning of sections and the IBR is eventually discarded.

The file loader and the file writer, during their execution, interact with two interpreters that operate at two different representation levels. The *executable interpreter* is the one which is directly responsible for building up the IBR for the relocatable file. When faced with instructions belonging to the code of the program, it gives controls to the *code interpreter*. The latter converts the machine-code instructions into an abstract representation which encodes the entire program logic as a chain of instruction descriptors. This representation is eventually fed back to the executable interpreter, which derives descriptors for functions and ultimates the IBR. In the inverse direction, the one needed by the file writer, functions and instructions in the IBR are converted back to their binary encodings.

It is worth saying that both the executable interpreter and the code interpreter are designed in such a way to support respectively multiple *object file formats* (OFFs) and multiple *instruction set architectures* (ISAs). Currently, Hijacker only supports ELF relocatable file formats and IA32/AMD64 machine-level encoding, although support for additional formats and instruction sets is on its way.

2.3.2 IBR

The Intermediate Binary Representation (IBR) provided by Hijacker can be seen as a pointer-based representation of the same contents included in the input relocatable file and referenced through displacements. Observe that manipulating memory pointers is much easier than manipulating displacements. For example, inserting a new instruction only requires to change the memory pointers of the pivot instruction and update those of the newly inserted one. Hence, changes are local and don't disrupt the value of other pointers. On the contrary, if the same insertion were performed directly on the relocatable input file, it would require to shift the displacements of all instructions that follow the newly inserted one, thus causing global changes.

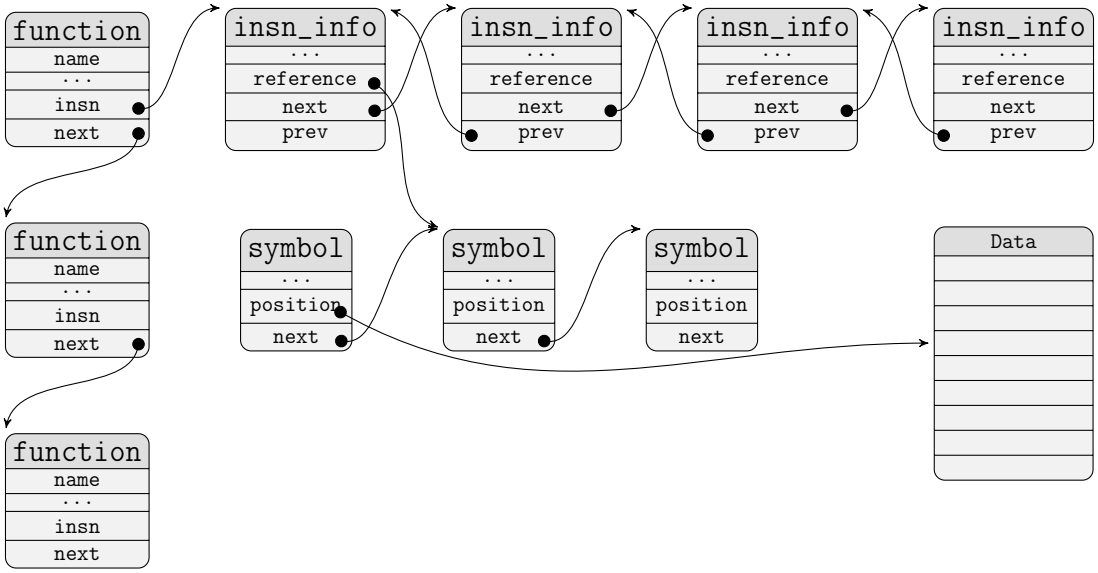

FIGURE 2.3: *Hijacker's internal binary representation*

The internal composition of the IBR takes much of its inspiration from ELF relocatable files. As such, every other supported object format will be converted into a representation that is highly inspired from ELF. In this representation, we find three different components: sections, symbols and relocations. Two additional types, functions and instructions, are built for convenience and find no direct equivalent in any relocatable OFF. They are required to provide Hijacker with a basic abstract representation of program code. A visual description of Hijacker's IBR is provided in figure 2.3.

Section

Sections represents coarse-grained portions of a relocatable file. Unavoidable ones are those for program code and program data. Other ones are self-referential and contain, for instance, symbols or relocations. As already explained, some of the sections that make up the file will eventually become segments which can be loaded in memory when a process is created. All sections that hold a portion of the compiled program in terms of code and data belong to that category. Other sections, such as those which are self-referential, are consumed by the static linker upon linking while producing the final executable for the program.

Symbol

In ELF and other OFFs, symbols represent finer-grained byte sequences than sections associated with a name, a size and a meaning in terms of

linking logic. They are the main target of relocations and can represent data object, functions, relocatable sections and other entities. Symbols are contained into a special section named symbol table. Inside the symbol table, each entry is a different symbol. Beside the symbol name, size and other flags, a symbol's entry also tells the linker the section and displacement within that section in which the symbol will be located inside the final executable file. This is true for data symbols representing object such as array and variables in general.

Relocation

Relocations are the peculiar feature of relocatable files. A relocation is an entry in a special-purpose relocatable section which describes that a reference to a given symbol is performed at a given point in the program (e.g., in the program code). Stated differently, a relocation represents the fact that at a given displacement from a section, a reference to a certain location displaced from the beginning of a symbol occurs. The most common forms of relocations are references to C variables—triggered by memory movement operations,—functions—triggered by call instructions—and instructions—triggered from jump tables in data sections to encode the addresses of `case` statements, but also the addresses of functions in the presence of indirect branch instructions.

Relocations are a convenient form of annotation for memory references whenever virtual addresses aren't available. Since references to symbols are expressed through rich meta-data, the exact location of such symbol in terms of its virtual addresses is not needed. By the time the linker concludes its job, all relocations have been consumed and flattened into real hard-wired virtual memory addresses which explicitly refer locations in the executable where symbols reside.

Instruction and functions

Instructions in Hijacker are represented with high-level descriptors that simply encode the size and type of the instruction (e.g. memory read, memory write, integer, floating point, logical), its displacement from the beginning of the code section, the presence of a relocation to a data symbol (if any), and other architectural details. Moreover, instruction descriptors are linked into a bi-directional chain that makes the implementation of many low-level instrumentation tasks such as instruction insertion or deletion much easier. Functions are derived from the instruction chain by checking the original displacements of instructions against the displacements of the respective function symbols. Eventually, a list of function is created where each function records mainly its name and first instruction.

2.3.3 Back-end

The back-end represents the core of the instrumentation facility. It is composed mainly of the *rule engine*, which is responsible for executing the previously stored rules on the current IBR. Many different rule types are supported. Among them, we can mention those that add new instructions to the representation or those that replace existing ones. Furthermore, Hijacker allows to easily inject new code into the program (either written in Assembly or C technology), as well as to transparently call user-defined function at desired point so as to execute the actual desired functionality (be it monitoring, profiling, debugging or whatnot form of logic). Finally, the *instrumentation engine* is invoked by the rule engine to actually manipulate the IBR. It is composed of many manipulation functions for section, symbol, relocation, function and instruction descriptors.

2.4 State of the art

In the rest of this chapter, an evaluation of some of the most popular instrumentation frameworks is provided. Later in this section, such frameworks are compared against each other and with the Hijacker toolkit. Hopefully, this should put into perspective the kind of instrumentation that I wish to achieve for my thesis, in terms of its strengths and limitations.

2.4.1 PEBIL

PEBIL (PMaC's Efficient Binary Instrumentation Toolkit for Linux) is a static binary instrumentation tool for Linux x86 and x86-64 which uses relocation as its main technique [13]. Not only that, but it is heavily oriented toward lightweight and general-purpose instrumentation, just like Hijacker. The high-level functioning of PEBIL is as follows. Whenever an instrumentation point is encountered while scanning the application's code, PEBIL injects a branch instruction that gives control to an instrumentation preamble. This instrumentation code saves the program state, invokes a user-defined function that performs the requested tasks, restores the previously-saved state and then branches back to the instrumentation point.

To provide sufficient space for the insertion of branches, PEBIL relocates code at the function-level so that before any possible instrumentation point a branch instruction can be injected. This procedure is split into four different steps. In the first step, the contents of original functions are relocated into an area of the code section that is allocated by PEBIL and private to it. Relocating the contents of function is an elegant way to leave function entry point unaltered. To link the entry point of the original instruction to

the actual relocated contents, the original entry point is replaced with an unconditional branch to the relocated function.

The motivation behind function-level relocation is that typical text segments in a final executable are such that the next instruction after the end of a function is the entry point for another function. This is to say that function code is not aligned nor padded, hence inserting additional instructions in the code would require shifting all those that come after the insertion point, included the entry points to other functions. This procedure is particularly delicate when it comes to executable files, as relocation entries are gone. Therefore, references to other instructions and data object must be recognized and update while shifting the instructions. Shifting is further complicated by the fact that indirect call instructions may be present in the code, which are hard to resolve before run-time. If PEBIL changed the entry points to functions, it would have to tentatively discover indirect calls, trace back call tables in data segments and update entry point addresses accordingly. Quite a complex and unreliable task.

The second step deals with padding instructions in the relocated code. For each instruction, an empty space is inserted before it and filled with a sufficient number of null instructions as to accommodate the potential insertion of a branch. On x86 and x86-64, 5 bytes are sufficient to hold a jump instruction to a near code location. Whenever a regular instruction is already a branch then it is 'self-padded', meaning that the byte length of a branch instruction taking less than 5 bytes is increase up to this size. Observe that this task would not be needed if the ISA of reference were that of a CISC architecture, since in that case all instructions would take up the same byte length. However, the cost of supporting a quite widespread family of processors such as x86-(64) is paid exactly by its intrinsic ISA complexity.

The third step is the actual instrumentation. If a regular instruction in the relocated code is recognized as an instrumentation point, a branch instruction is inserted before it—or in place of it when it comes to existing branch instructions. By executing the newly inserted branch, control is passed to PEBIL's instrumentation preamble. As already explained, this preambles saves the current architectural context and then invokes an analysis procedure which is defined by the user in a high-level language such as C. After returning from the analysis procedure, the architectural stare is restored and the application execution is resumed.

The major overhead of PEBIL's instrumentation at run-time comes from the insertion of additional branches in the code to achieve function-level relocation and invoke the analysis routine, resulting in possible control hazard at the micro-architectural code level. Moreover, as explained by the authors, "since the code is being reorganized and expanded, some positive alignment and size optimizations that the compiler might have made on the

instructions in the function might be destroyed." Nevertheless, the effects of relocating code and padding instructions alone, that is without branching to PEBIL's instrumentation code, are measured on the SPEC CPU2000 Integer benchmark to be no higher than 5% more of the non-instrumented execution.

Another critical source of overhead comes from the analysis procedure itself, whose execution can pollute instruction and data caches to an arbitrary extent. Not only that, but it also requires PEBIL to save and restore the processor context, which is a very expensive operation. Notice that beside functions written in a higher-level language, PEBIL also enables the insertion of lightweight assembly-level instrumentation snippets. Code snippets written in low-level language minimize the effects of cache pollution and relieve PEBIL from the save/restore logic. The task of preserving the value of registers is now in charge of the snippet itself, which can optimize it by only saving and restoring those registers that will be actually employed.

An experimental evaluation of PEBIL's performance is derived, again, from running an instrumented version of the SPEC CPU2000 benchmarks. The analysis function in this experiment is in charge of counting the number of times that a basic block is executed. Basic blocks are explained to a deeper extent in chapter 4. Basically, they are contiguous sequence of instructions where either the first instruction is the target of a jump, or the last instruction is a jump operation itself. Counting the number of times a basic block is executed allows to estimate the overhead introduced by PEBIL while minimizing that introduced by the analysis code, which can be implemented as an Assembly snippet. As can see from picture 2.4, the average overhead measured for PEBIL is 62%, compared to 151% for Pin, 408% for DynamoRIO and 734% for Valgrind.

2.4.2 DynamoRIO

DynamoRIO is defined by its authors as a dynamic optimization infrastructure for the IA32 family of architectures and the Windows environment [14]. The main idea behind this tool is to optimize the execution of a program while it is running, thus overcoming the obstacles that are traditionally posed by static compilation. Compilers are in fact incapable of predicting the behavior of applications at run-time, therefore the kind of optimizations that can be performed on program is restricted to those enabled by static analysis. The main obstacle to dynamic optimization is that of maintaining control over the running application at frequents points in its execution. In practice, this means intercepting instructions that divert the regular execution flow of the application—for instance, branch instructions—as well as instruction that perform a more aggressive form of control flow hijacking—

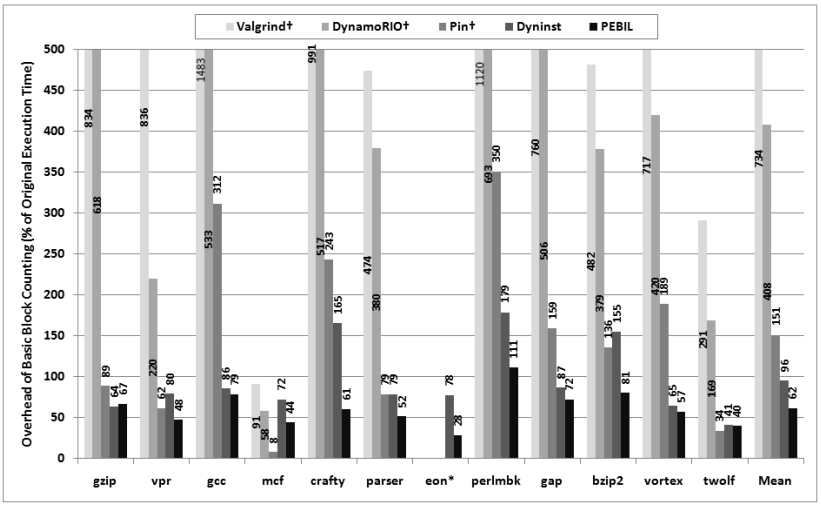

FIGURE 2.4: *Performance evaluation of PEBIL with respect to other instrumentation tools.*

such as those raising exceptions or invoking privileged services.

Conceptually speaking, DynamoRIO is a JIT optimizer which acts as a wrapper around the actual executable. Code from the original program is executed natively in blocks of consecutive code called *fragments*, which terminate as soon as a control flow instruction is met. Such instruction is intercepted so as to give control back to the DynamoRIO VM. Fragments persist in a software-defined code cache for fast retrieval and are linked together to avoid returning to DynamoRIO's code at boundaries whenever possible. Among them, those that have been identified as being executed more frequently can be optimized by the VM to improve the performance of the running application. Observe that beside direct and indirect branches, Windows supports several other abnormal ways to transfer flow from one point in the program to another. For instance, Windows supports callback mechanisms for thread-level message queues. Another form of control flow hijacking is exceptional control flow. All such mechanisms are successfully intercepted by DynamoRIO.

To evaluate the slowdown incurred by applications run through DynamoRIO, the SPEC2000 suite of benchmarks is executed without any form of dynamic optimization. The results of this experiments in picture 2.5 show that apart from pathological cases the slowdown introduced by DynamoRIO is never higher than 2.0x.

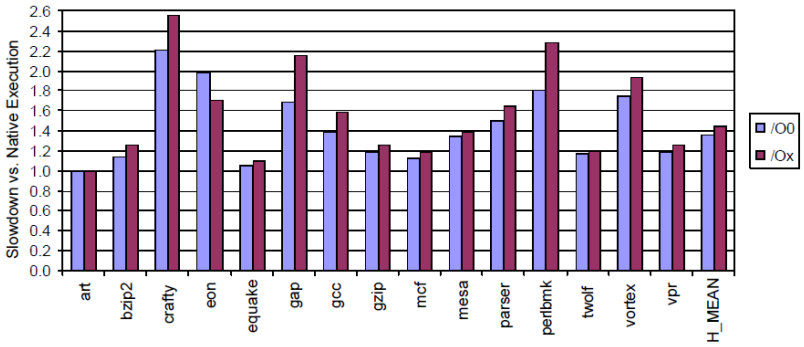

FIGURE 2.5: *Performance evaluation of DynamoRIO with no dynamic optimization.*

2.4.3 Pin

Pin is a dynamic instrumentation framework available for Linux platforms for IA32, AMD64, IA64 and ARM architectures [15]. Its design goals are usability, transparency, portability, efficiency and robustness. The first three goals are achieved thanks to a flexible API to develop Pin plug-ins, called *Pintools*, which allows users to instrument arbitrary code locations. Many of the tasks needed to preserve the correctness of the application—such as saving and restoring the processor state—are handled in a transparent manner by Pin, which also abstracts away all the architectural-specific issues that must be handled. Efficiency is achieved by improving JIT instrumentation through code caching, trace linking, inlining of analysis functions and register re-allocation. Another important source of efficiency is given by the Pin's ability to attach and detach from a running process, so as to activate and deactivate instrumentation. Lastly, the robustness of Pin comes from the fact that by using dynamic instrumentation many classic issues of static approaches—such as not knowing the actual targets of indirect branch targets or not being able to instrument external libraries are—are naturally absent.

The main technique adopted by Pin to instrument code is JIT native execution. Pin receives the original executable as an input, intercepts the first instruction and decompiles a contiguous sequence of instructions, named *trace*, which ends with a branch instruction. Once a trace has been identified, Pin re-compiles it in such a way as to (a) instrument it according to the desired rules and (b) make sure that the last instruction of the block gives control back to Pin's code. Then, Pin executes this re-compiled block natively on the architecture. It must be noted that for efficiency purposes, all code which passes through Pin is transformed from one ISA directly into the same ISA without passing through an intermediate format. After regaining

control, Pin repeats the same step for a trace rooted at the original branch target. A software code cache is maintained to store previously-recompiled traces so as no to incur again in the recompilation overhead.

Pin also supports trace linking, in an attempt to directly move from one trace to another whenever the destination trace is already in the code cache. This avoids the expensive passage from the application code to Pin and from Pin back into the application code. Another useful performance booster is the inlining of analysis functions inside the code to be instrumented, similarly to what PEBIL does. Without inlining, control must first pass to a bridge routine which protects the current architectural state prior to calling the analysis snippet. In total, instrumenting code through the bridge routine requires two calls and two returns. With inlining, on the contrary, the bridge routine is avoided and the analysis snippet is directly inserted into the application code, thus sparing the previous instrumentation cost in its entirety. A register allocator is in charge of handling possible register conflicts that occur in the analysis routine, by exploiting possible dead registers or temporary spilling out-of-context application registers to the stack.

Experimental results on the SPEC2000 test suite indicate that the speed-up gained from running Pin without any optimization to running it with all optimization enabled can be as large as 10x on some benchmarks, as seen in 2.6. A comparison of Pin, DynamoRio and Valgrind on the basic block counting experiment suggests that Pin is 2x faster than DynamoRio and 3x faster than Valgrind. On the other hand, when instrumentation is disabled to appreciate the intrinsic overhead introduced by these tools at run-time, Pin is considerably faster than Valgrind, but slightly less performing than DynamoRIO. This is likely due to the fact that while DynamoRIO is thought as a framework to optimize the *execution* of applications, Pin optimizations are thought for *instrumentation* purposes.

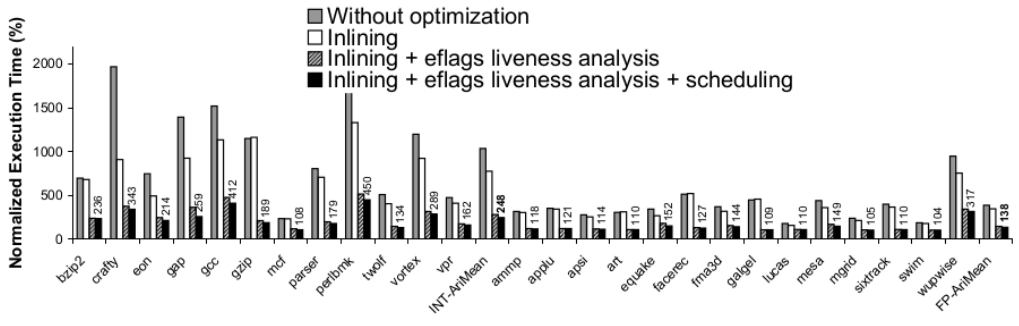

FIGURE 2.6: *Performance evaluation of Pin alone with and without optimizations.*

2.4.4 Dyninst

Dyninst is a binary instrumentation tool which puts strong emphasis on the flexibility goal [16]. It can instrument a binary file either by using static instrumentation techniques or by relying to dynamic approaches (a technique called *anytime instrumentation*). Moreover, it can instrument executables at any level of abstraction (*anywhere instrumentation*), ranging from instruction-level to function-level, including basic blocks in the control-flow graph (for an explanation of these two concepts, see chapter 4). The authors of Dyninst claim that it able to produce an instrumentation overhead which is always proportional to the number of instrumented locations, independently on the kind of instrumentation being used.

The basic idea for anywhere instrumentation is to allow user-defined analysis functions to manipulate program code by providing different kind of instrumentation points. Typical instrumentation tool, included Hijacker, only allows an external function to gain control over an instrumented application prior to or immediately after the execution of a particular kind of instruction (e.g., memory read, memory write, floating point, etc.) Dyninst, on the contrary, provides additional instrumentation points such as function entry/exit, basic block entry/exit, loop entry/exit and loop iteration entry/exit. These additional points are supported through the manipulation of a global CFG whose nodes and edges are appropriately tagged. A peculiarity of Dyninst is that instrumentation code is directly inserted in this global CFG by means of additional nodes and edges. The augmented CFG which results from the instrumentation is later used to generate the final instrumented binary code.

To achieve proportional instrumentation cost, Dyninst chooses the code that must be relocated and patched, then creates a relocated copy for the selected code and instruments it with the desired rules. Finally, it patches the original code so that control will be diverted to the relocated region. To achieve anytime instrumentation, two techniques are introduced. The first, called *state interception*, allows to move to the relocated region while running the original code version. Rather than patching the original code, state interception overwrites the process context by altering the content of the program counter, so that control is transparently passed to the instrumented version. The second technique, named *iterative instrumentation*, allows users to augment previously instrumented code with further instrumentation, or to remove pre-existent instrumentation so as to switch back to normal execution. It internally employs state interception to move back and forth between instrumentation versions, but it also allows to achieve overhead proportionality since non-instrumented code can always be executed.

To evaluate the performance of Dyninst, the SPEC2006 benchmark was

instrumented with a simple basic block counting logic. The obtained results suggest that the overhead incurred by Dyninst is competitive with that of Pin, DynamoRIO and PEBIL on almost all benchmarks. Moreover, Dyninst achieves considerably less overhead than Pin when it comes to measuring the proportionality metric, as shown in 2.7. This results from the fact that Pin unconditionally intercepts all the executed code, included non-instrumented one.

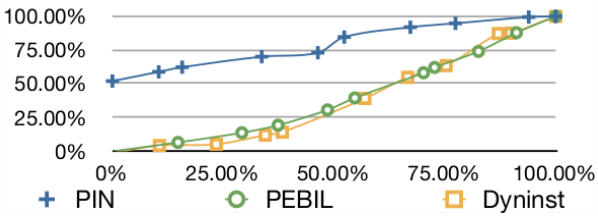

FIGURE 2.7: *Evaluation of the proportionality metric for Pin, PEBIL and Dyninst.*

2.4.5 Valgrind

The main purpose of Valgrind is to enable extremely flexible and powerful dynamic instrumentation which, although heavyweight, allows to carry out complex analysis task which would not be feasible with other instrumentation tools [17]. The flagship of Valgrind, as opposed to other tools, is the support for shadow values. A shadow value is an in-memory annotation that describes the value of one register of memory location used by the program with another value. The meaning of a shadow value is defined contextually to the kind of analysis one wishes to perform. Apart from that, the execution of a program is always reflected by a shadow execution which operates on shadow values.

Many popular plug-ins have been implemented on top of Valgrind's core and its support for shadow values. The most popular tool, Memcheck, uses shadow values to keep track of which bit values are uninitialized and can report accesses to undefined memory area with extremely high precision [18]. Other useful tools have been developed, but overall they account for less than 20% of Valgrind use by programmers. Another reason for running applications against a Valgrind tool is that Valgrind has its own memory allocator that is able to intercepts calls to `malloc` and `free` operations. This allows Valgrind tools such as Memcheck to even notify the user about incorrect usage of the heap, which is mostly due to memory leaks.

Similar to other dynamic instrumentation frameworks, Valgrind uses dynamic binary re-compilation. Whenever a program is run in Valgrind, it is

loaded into the same address space as the invoked Valgrind tool. Application code is disassembled and re-compiled one code block at a time. The disassembly product is an IR which is the target of the instrumentation tasks carried out by the currently executing Valgrind tool. After instrumentation, the result of is stored in a core cache so that the same code block doesn't have to go through the same process again in the future. Whenever the execution of a translated block completes, the application relinquishes control and gives it back to the Valgrind dispatcher, which checks whether the next block to execute is already present in the code cache and, if not, creates it. Contrarily to other dynamic instrumentation tool, direct chaining of code blocks is not supported.

In the IR, the actual registers used by the host machine (hence called *host registers*) are replaced with a set of *guest registers* to abstract away the architectural limitations of having high register pressure and frequent register spilling. Furthermore, a single architectural instruction might be replaced by multiple IR instructions that describe, beside the visible effect of the real instruction, all its side-effects. This IR is convenient to manipulate for Valgrind tools because it is architecture-independent and highly expressive.

To translate a single code block into an instrumented block, the process is as follows. First, a code block is recognized according to some criteria (e.g., by parsing instructions until a conditional branch is met). Then, each instruction is disassembled into its IR code, which operates on guest registers. The resulting IR is optimized by removing redundant and unnecessary operations, as well as performing constant folding, common subexpression elimination and other kinds of flattening analysis. Then, the Valgrind tool instruments the code block according to its own policy. Eventually, the IR is translated back into machine-dependent code by first replacing guest registers with host registers and then converting each IR expression into a valid machine-level instruction.

The experiments conducted on the SPEC2000 benchmarks to evaluate the baseline overhead induced by Valgrind yield results which are consistent with the findings of other researchers. Valgrind is 4x slower than Pin and 4.4x slower than DynamoRIO when no instrumentation is performed. Similarly, when basic block counting is evaluated, the slowdown with respect to Pin and DynamoRIO are respectively 3.3x and 2.0x. However, the purpose of Valgrind is to enable heavyweight complex instrumentation, rather than lightweight instrumentation tasks. For this reason, the observed slowdown as compared to other instrumentation tools is actually expected and quite acceptable in perspective.

2.4.6 Comparison

This section is devoted to comparing the previously-discussed frameworks according to the following criteria. First of all, tools can employ either static or binary instrumentation. The benefits and drawbacks coming from the employment of one technique over another have been discussed in section 2.2. Second, they can work at different stages in the compilation process, with the differences being highlighted in section 2.3. Another interesting classification is the level of granularity at which instrumentation can occur (e.g., function-level, instruction-level, etc.) which is reflected in the overall flexibility of the instrumentation toolkit.

Beside the usage of static or dynamic approaches, an instrumentation framework is also identified by the specific techniques that are put into practice (e.g., relocation or native IR-aided JIT instrumentation). Among them, we stress multi-versioning, which in principle can be employed to switch at run-time between differently instrumented version of the same executable (including the baseline non-instrumented version) so as to place different degrees of overhead over the instrumented application. This can be useful in certain scenarios (e.g. transactional processing systems) where the workload to which an application is subject can change over time as a function of external input or system events.

As reported in table 2.1, Hijacker holds a unique spot within the spectrum of instrumentation tools. It is a static instrumentation framework like PEBIL and also supports the same family of architectures. However, it works on relocatable files, while PEBIL instruments executables. As such, Hijacker can grants itself the luxury of using inlining without disrupting any references from code to data and viceversa, thanks to relocations. Moreover, as opposed to PEBIL, it has a rudimentary IR support for code and is highly oriented toward multi-versioning. The remaining tools, being all JIT VMs, can too be easily compared. Pin and Valgrind support the higher number of architectures, while Dyninst stands out due to its support for anywhere instrumentation (arbitrary granularity) and anytime instrumentation (which entails multi-versioning). DynamoRIO is interesting because is oriented toward efficient execution, while Pin makes extensive use of register re-allocation techniques to minimize run-time instrumentation overhead. Lastly, Valgrind is awarded honorable mention due to its extremely peculiar support for shadow values.

-	PEBIL	DynamoRIO	Pin	Dyninst	Valgrind	Hijacker
OS	Linux	Windows	Linux	Linux	Linux, OSX	Linux
ISA	x86-(64)	x86	x86-(64), IA64, ARM	x86-(64)	x86-(64), ARM	x86-(64)
Approach	Static	Dynamic	Dynamic	Both	Dynamic	Static
File type	Exec	Exec	Exec	Exec	Exec	Reloc
Granularity	Instr	-	Instr	Any	-	Instr
Techniques	Relocation	Native	Native	Relocation	Native	Inlining
IR	No	No	No	Partial	Yes	Partial
Versioning	No	No	No	Yes	No	Yes
Other	-	Run-time optimization	Register re-allocation	-	Shadow values	-

TABLE 2.1: *A comparison of binary instrumentation tools*

CHAPTER 3

Memory Tracing

There is nothing like looking, if you want to find something.
You certainly usually find something, if you look,
but it is not always quite the something you were after.
— J.R.R. Tolkien, The Hobbit (1997)

Memory tracing is the fine art of collecting the stream of memory addresses generated by an application, also called *memory access pattern* or *memory access stride*, throughout its execution. The output of this activity is a *trace*—i.e. a sequence—of memory address that have been touched at run-time, since the beginning of a program execution or the closest previous processing point. For instance, accesses can be buffered for a given period of time or buffer size, beyond which the in-memory storage is flushed and the buffer is consumed by an external monitoring tool. Once reset, the buffer can accommodate a new wave of accesses. An alternative technique, perhaps less used, is to consume accesses in a synchronous way as a stream.

Memory tracing is typically a means to a higher end and is used both by computer architects and software engineers. In the first case, memory address traces are generally observed with the purpose of improving the architectural design of an existing memory subsystem, either in terms of performance or energy consumption. In the second case, traces are collected with the intent of making better employment of the architecture in a given application (e.g., by improving the cache hit ratio through cache-oblivious algorithms). In either case, converting collected data into actual information is rarely the main issue. More often, the act of collecting data itself is the real hurdle.

When it comes to memory access tracing, the intrinsic difficulty of this task can be decomposed into two complementary issues: (a) how many and which accesses to collect; (b) when the consumption actually occurs. To this extend, we now proceed to discussing two common tracing optimization. The former is related to trace collection, the latter to trace consumption. Both techniques, perhaps unsurprisingly, are typically based on the usage of static or dynamic instrumentation. As a matter of fact, memory tracing

can be seen as the most classic application of instrumentation to the art profiling. In the remaining of this chapter, while talking about tracing optimization techniques and state-of-the-art memory tracing tools, we will often assume the exploitation of instrumentation techniques.

3.1 Sampling

When it comes to tracing problems, it is important to understand the trade-off that exists between *tracing accuracy* and *tracing overhead*. Suppose we have a very memory-intensive application with a quite huge trace. Since this trace contains all accesses performed by the application, it is 100% accurate, thus making every form of downstream analysis thorough and reliable. However, collecting such traces could pose a big time and space overhead on the profiling tool that performed this task, possibly with unacceptable results.

For this reason, tracing is usually implemented through sampling techniques that collect a 'relevant' subset of the memory accesses ever performed by the application. Perhaps not surprisingly, the exploitation of sampling doesn't reduce the complexity of tracing, but in fact moves it to the issue of finding a relevant subset. Formally speaking, *sampling* acts as a filter on a set of memory accesses, possibly coded into a raw format and not directly recognizable, as would happen with the machine-level representation of a program. Program code is parsed instruction by instruction to detect memory access operations. Upon the occurrence of a memory access, an *heuristic function* is run that decides whether such access must be collected or not. The complexity and effectiveness of such heuristics depend on many factor that will be later explained. As a result of this filtering, a subset of accesses is produced that includes all and only the sampled operations—that is, those that the heuristic function decided to collect.

Clearly, the higher the number of memory accesses that are collected, the higher the accuracy of the produced trace and the higher the overhead. Collecting 100% of accesses yields full tracing and, as already said, is a show-stopper because the benefits of increased accuracy might not compensate for the increased overhead (a phenomenon called *over-sampling*). On the other hand, sampling a very low fraction of memory accesses will yield extremely low overhead, to the detriment of accuracy which may be disappointing (*under-sampling*). Finding the optimal trade-off between accuracy and overhead is the most difficult problem, but undoubtedly also the most exciting one.

3.1.1 Static sampling

Typically, sampling can be performed statically or dynamically. *Static sampling*, is a form of sampling performed at compile-time. It usually relies on control-flow analysis (CFA) as well as data-flow analysis (DFA) algorithms to recover the highest amount of semantics from the machine-level representation of a program. At this level, as will be better explained in chapter 4, control flow is represented as a graph with multiple destinations for a single place in the code. Because of this high degree of uncertainty on the actual execution path taken by the application, static sampling heuristics tend to be extremely complex.

For static sampling, a logical way to proceed is to analyze a small subset of the code regions that account for most of the execution time of an application. Finding the 'hottest' regions is not necessarily easy and the exact way to do that depends on our definition of hottest. Clearly, it must be a region that will be visited 'often' at runtime. An example of smart static sampling can be found in [19]. At that point other decision must be made to decide which accesses to collect within the same region, with the purpose of building a relevant trace.

The big advantage of this approach is that these functions have at their disposal a complete representation of the program, therefore can span big code regions and improve their decisions by reasoning on the program on a global scale. Also, they can result into little overhead at run-time.

3.1.2 Dynamic sampling

Dynamic sampling, as opposed to static sampling, occurs at run-time. Just like dynamic instrumentation, it typically produce higher overheads. However, such overheads being equal, it also yields generally higher accuracy. The execution path taken by the application is observed as it evolves. For this reason, dynamic sampling heuristics tend to be simpler and more reasonable—they work on the actual memory access pattern exhibited by the program. On the other hand, they only have at their disposal a more restricted view on the program, devoid of any unseen past or sibling state [20].

For example, to leverage on the previous idea of 'hottest' code regions, an dynamic approach might decide to count the number of times that a region is executed, then decide to analyze only those re-occurring regions whose with an access count that exceeds a certain threshold. More convoluted form of learning can be employed, depending on the downstream analyze that we wish to carry out.

3.2 Buffering

Consumption of traces, be it sampled or full, can occur according to two different temporal schemes, namely synchronously or asynchronously. The first case, the naive one, expects all accesses to be collected as soon as they materialize in the execution of the program. It is suitable whenever we require a timely stream of accesses that must be consumed on the fly. However, albeit straightforward, it tends to pose excessive overhead while profiling the application. For this reason, a generally better form of tracing involves *buffering*. It entails storing the collected trace into a temporal storage that is later 'flushed' to the consumer, at an appropriate moment in the execution of the application—possibly even outside of its critical path, e.g., through a background consumer process.

The natural justification to this asynchronous consumption scheme is that most of the accesses need not be consumed on the fly. In the long run, profiling tools need to collect as much information as possible about the running process, but not immediately after raw data materializes. While this increases the 'time-to-processing', which is the delay experienced by a single datum before being consumed by the profiler, this also allows to decrease the overall profiling time for the entire application. Observe that the cost of processing a single access consumer-side tend to be comparable to that of processing a much wider range of accesses at once. Therefore, buffering is a technique which can drastically improve throughput to the detriment of latency and is suitable in all those profiling contexts where time-to-processing is not a interesting metric.

3.3 State of the art

Much of the interesting results obtained in the field of memory tracing are related to performance prediction of programs. Therefore, in the following section I will focus on performance prediction frameworks.

3.3.1 MetaSim Tracer

Among them, a trace-driven memory simulation tool called MetaSim Tracer [20] is oriented to general scientific applications running on arbitrary HPC platforms. It simulates how a given application will interact with the memory subsystem of a target machine. At the end of the simulation, statistics on memory and communication access patterns are collected that enable further analysis. The actual simulation is based on collecting *application signatures*—like expected cache hit rates and memory operation counts—as

well as *machine profiles*—which information on the cache sizes, associativities, line sizes and other features of the target machine. These two datasets are collected independently one from another and then combined through a convolution method to predict the *memory execution time* (MET) of that application on that machine.

Apart from the convolution method, MetaSim Tracer is interesting because it can speed up address traces collection by using dynamic sampling. As explained by the authors, the basic idea of sampled tracing is "to turn the trace collection of an application on and off at certain intervals, while running the application". As a result, only a small percentage of the addresses are collected, which depends on user-specified parameters such as interval length and number of contiguous traces to collect since the beginning of the interval (called *sample size*).

On top of this basic sampling mechanism, two optimizations are built that revolve around the notion of minimal code regions. A minimal region is a contiguous sequence of instructions that cannot be divided into two smaller regions without causing an exclusive control flow connection between the first one and the second. It will be best explained in chapter 4. However, in principle, for two connected minimal code regions the source and the destination must respectively have an in-degree and an out-degree greater than one.

The first optimization poses an upper limit on the number of times a minimal region is processed to collect traces. This limit is justified by the fact that a single minimal code region exhibits similar behavior across multiple passes in the same program execution. For instance, the average cache hit rate over 1'000 traversal on a given code region cannot deviate much from that averaged over 1'000'000 traversals. Another optimization entails tracing only certain regions of the application, usually the hottest ones—i.e., those that are executed the most. Again, the motivation is that in many applications, "there are only a small number of regions that account for most of the wall-clock time in the application [and] capture most of the [...] performance attributes." To determine which region to trace, two static passes over the application are performed, the first one being that which identifies at run-time the hottest regions.

These trace time reduction methods are evaluated on an individual basis first, then combined to prove that they are indeed lightweight and accurate. The average slowdowns over the LINPACK benchmark are of 152x for 10% sampling, 130x for an upper sampling limit of 1'000 times, 154x for sampling the top 20 hottest basic regions. The respective relative errors in prediction accuracy are 3.8%, 22.2% and 12.1%. Tables 3.1, 3.2 and 3.3 provide individual performance and accuracy evaluations respectively for the periodic sampling criteria, the upper limit on block traversals and the

hotness of blocks. A combined experiment involving all these techniques (1% sampling, basic-block limit of 200 and top 100 basic blocks) yielded a slowdown of around 7x with percentage errors falling below 4%.

Sampling size	Slowdown factor	Error
100%	859	4.6%
10%	152	-3.8%
5%	141	-4.6%
1%	132	-8.4%

TABLE 3.1: *MetaSim Tracer collection time and prediction accuracy comparison using trace sampling.*

Basic block limit	Slowdown factor	Error
None	859	4.6%
10'000	134	-20.1%
1'000	130	-22.2%
100	130	-23.4%

TABLE 3.2: *MetaSim Tracer collection time and prediction accuracy comparison using an upper sampling limit on block traversal.*

Basic block hotness	Slowdown factor	Error
All	859	4.6%
Top 20	154	-12.1%
Top 10	151	-9.6%

TABLE 3.3: *MetaSim Tracer collection time and prediction accuracy comparison using basic block hotness.*

Contrarily to this work, my thesis deals with static sampling. It defines a measure of hotness for minimal regions on top of the CFG that simply counts the number of cycles to which a single region participates to. Time-based sampling (i.e. turning on and off sampling according to an interval) would require dynamic sampling, hence dynamic instrumentation techniques, that are outside the scope of this thesis. Notice that the thresholded sampling method is too a time-based form of sampling, since it simply defines that a region be not scanned again when its dynamic counter reaches a desired threshold.

Another work on the same simulation tool [19] improves on the previous one through the adoption of an instrumentation toolkit for the Alpha

platform called ATOM [21]. The main idea of this work is that the major slowdown doesn't come from instrumenting the program, but rather from simulating the interaction of the application's address stream against the cache. As a result, the main purpose of the paper is to discuss a way to defeat this slowdown by reducing the number of instructions that must be simulated without impacting (much of) the simulation accuracy. The improvements are obtained by using techniques developed in another simulator, called SimPoint [22].

There are three main ways to perform simulation of multi-threaded program driven by memory traces:

- Full-trace simulation (MetaSim Tracer)
- Simulation using periodic simulation intervals (MetaSim Tracer)
- Simulation using (aperiodic) intelligent simulation intervals (SimPoint)

Full-trace simulation is extremely accurate but performs way too poorly with respect to the actual execution time of the program. In contrast, the authors explain that by using regularly-spaced intervals the number of instruction that must be simulated can fall below 10%, while retaining almost full accuracy. Unfortunately, this still yields a two orders-of-magnitude slowdown compared to regular execution. According to the experiments conducted in the paper, even 1% regular sampling produces at least one order-of-magnitude slowdown at the expense of lower accuracy.

Evaluation is carried out by comparing intelligent simulation, as performed by SimPoint, against full and periodic 10% and 1% simulation. The former is conducted by SimPoint using light-weight dynamic basic block traces and feeding an off-line classification algorithm that establishes the weight of each simulation interval (made by multiple contiguous basic blocks). The accuracy of simulation is evaluated by computing the cache hit rate at basic-block level, weighted by the frequency of execution of the basic block in the full trace (for cases 1. and 2.) vs. frequency of execution in the intelligently-sampled trace (for case 3.).

The results, depicted in figure 3.1, suggest that SimPoint-guided cache simulations are about as accurate as 10% periodic trace for L1 cache hit rate. The overall cache hit rate differ from that of a full-trace simulation by an average of 0.25%, while that of 1% and 10% periodic traces is off by an average amount of 0.43% and 0.07% respectively. On the other hand, SimPoint-guided simulation is faster than even 1% periodic sampling-based simulation and much faster than the 10% case. Observe that a relevant limitation of this paper is that experiments are run on benchmarks where each process exhibits homogeneous behavior, therefore it is sufficient to learn the right simulation intervals for a single process to know those of

all running processes. On the other hand, we target general-purpose multi-threaded applications where execution dynamics (and therefore memory access patterns) can be different from thread to thread.

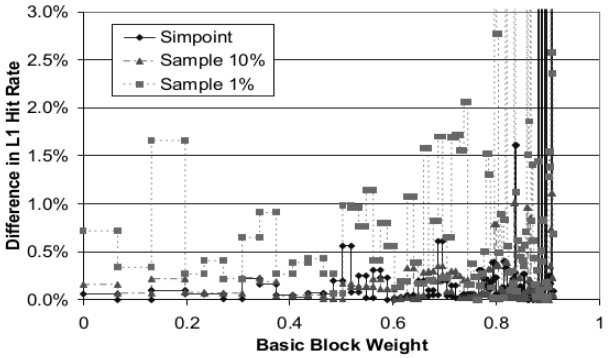

FIGURE 3.1: *Evaluation of the cache hit rate estimation error produced by SimPoint-based sampling.*

3.3.2 Effects of binary instrumentation on tracing

Continuing our discussion of trace-driven simulation tools that employ instrumentation-based techniques for trace collection and processing, [23] discusses the main sources of overheads coming from binary instrumentation. These overheads can negatively affect the simulation of program even when interval-based sampling, or even guided sampling like that enabled by SimPoint, is in place. The key factors that cause an instrumented binary application to run longer than the original one are:

1. Instrumented code executes more instructions that does the original binary. This cost is unavoidable and cannot be measured independently from the others to follow.

2. Jumping to an analysis routine snippet is a control flow interruption. The practical effect when analysis snippets are inserted after every memory instruction in an otherwise well-scheduled binary is that the pipeline frequently bubbles or breaks. The measured lower bound of this overhead is a 10x slowdown, resulting from having an analysis snippet that receives no parameters and does nothing.

3. Program state has to be saved when jumping to analysis snippets. This overhead can be quite expensive if the processor has many registers to save and later restore. Theoretically, instrumentation tools could target only the registers that are effectively used by the analysis

routine, but in practice many instrumentation APIs simply save all registers. The lower bound slowdown of this overhead when register saves are optimized, is of about 20x when using a one-line analysis snippet.

4. Analysis snippet pollute architectural caches. This is due to the fact that instrumentation code is interleaved with instrumented code, therefore there could be contention both in instruction and in data caches. This too can impact performances considerably, with slowdowns in the order of 10000x.

To optimize factors 3 and 4, buffering of memory addresses can be implemented. Collection happens upon accessing a memory location, but processing is done in batches, that is it is delayed until the current buffer is full. This check belongs to an analysis snippet located at the end of each basic block, executed much less frequently due to its nature (especially when blocks are large). Factor 3 is improved because state saving occurs only when the buffer gets full. This approach also reduces the frequency of interference with the program's cached data. It cannot improve factor 2 because an analysis routine is still required to synchronously collect traces.

To address the issue of control flow interruptions, the authors devised a technique which effectively reduces the number of times the effective address information is stored in a buffer, without affecting accuracy. Interestingly, this is achieved using static application analysis, and relies on two methods: chaining and delaying. The first technique group together memory instructions in a given basic block which share a common base register, the second generate effective addresses from base register values and immediate displacements by deferring the recording of these values and the actual address computations until the end of the basic block.

- *Chaining*: Memory instructions are grouped into chains, based on their addressing modes and the registers used to make up the effective addresses. All instructions with the same addressing mode, employing the same registers and having the same registers' values are grouped into the same chain. As a result, they only differ in their immediate displacements from the base address. Only one instruction in each chain is instrumented; the others' effective addresses can be calculated from the leader's address by storing just the residual displacement relative to that of the leader instruction.

- *Delaying*: Rather than storing the effective address of the leader instruction, register values are recorded in a deferred fashion wherever possible. These values can be then used in conjunction with static

knowledge of the basic block binary code, such as immediate displacements, to re-create the effective addresses of each memory instruction. The goal of delayed instrumentation is therefore to find a way to merge all memory address collection into a single instrumentation call at the end of the basic block. If the value of a register changes before reaching the tail instrumentation call, these changes are reversed to the correct value prior to computing the effective addresses (it only supports addition and subtraction).

The outcome of these optimizations is that slowdown can be improved from a factor of around 30x to 50x to a value of 5x on average on real applications (especially those with long running times). It's worth saying that in the authors' words, "[this] static analysis techniques will vary in impact based on the specific application to be traced", i.e. depending on how effective chaining and delaying can be in reducing the effective number of instrumentation points in hot basic blocks.

Similarly to the previous work, [24] describes ALITER, a lightweight instrumentation tool that employs deferred invocation of the user's routine through buffering of interesting events. Control is transferred to the user's analysis routine only when the buffer gets full. Furthermore, buffer maintenance is inlined into the application's code, thus avoiding an expensive call to a maintenance routine. Overall, these optimizations allow to counteract the effect of control hazards and cache pollution. The slowdown of acquiring address traces in ALITER never exceeds the factor 2x for the NAS benchmark, compared to those of other lightweight (but synchronous) tools such as ATOM and PIN, which are 20 to 40 times higher.

3.3.3 A threading-model for memory tracing

The last work discussed in this section is intimately related to instrumentation [25]. It, once again, explores memory tracing as a way "to understand the low-level interactions between a data intensive code and the memory subsystem of a multi-core processor or many-core co-processor." It augments the PEBIL instrumentation tool for Linux x86 and x86-64 with multi-threaded application support. This novel threading model is then compared by the authors to that of two other tools, Pin and Dyninst (see chapter 2).

In summary, each thread has its own private pool of instrumentation data structures, whose address can be retrieved in a hash-table based on the thread's ID and a hash function. The ID of each thread is obtained by querying thread-level meta-data and, in case of collisions, the execution is interrupted and it can be re-started with a bigger hash-table size. Address traces are collected into this thread-local memory area. A number

of optimizations are described, which decrease the overhead incurred while executing the statically-instrumented program:

- *Address trace batching*: Address traces are buffered and then consumed at once in batches to reduce the actual execution overhead.
- *Private pool's address caching*: Rather than re-computing the location of thread's private memory each time a new memory access must be recorded, PEBIL attempts to cache it into a dead register (if existent), so that subsequent instrumented accesses already have the value available into that register. This is done on a per-function basis and, whenever a dead register is found, re-computation of the memory pool location is performed only at the entry and re-entry points of a function.
- *Interval-based sampling*: Only a fraction of the memory accesses is actually instrumented (sampling). When this fraction is contiguous, execution can be seen as split into intervals and the instrumentation tool can disable and later re-enable instrumentation on a per-interval basis. In PEBIL, this is done by dynamically replacing instrumentation stubs with no-operations, while Pin and Dyninst can actually remove instrumentation code. Intervals have fixed size—1 billion instructions, while the program is running—and only the first 1% and 10% of the instructions of each interval are sampled in the experiments. This drastically reduces execution overhead, "while still allowing the properties of the memory address stream to be ascertained with an acceptable level of fidelity."

To measure the effects of memory address tracing only—that is, without processing—traces are buffered and then discarded as soon as the buffer becomes full. The results of these experiments, reported in 3.4, show that at a 10% sampling rate, PEBIL incurs an average slowdown of 2.9x compared to 4.4x with Pin and 897x with Dyninst. On the other hand, the overhead of collecting full traces for multi-threaded programs is higher in PEBIL (7.7x) than in Pin (4.7x), both of which are significantly lower than Dyninst (again 897x).

The paper is perfectly aligned with my purposes of tracing memory accesses in multi-threaded applications using static binary instrumentation. However, in practice it gives little information on sampling (i.e. tracing a subset of the accesses) as well as little original contribution to the sampling vs. accuracy trade-off. It's much more oriented to defining a reasonable threading model for instrumentation. Notice that it is not clear as to how sampling is actually implemented in these tools. Assuming that thread-level code is shared among threads, a crucial aspect is that instrumentation

Tool	Sampling	Average slowdown
PEBIL	100%	7.7x
PEBIL	10%	2.9x
PEBIL	1%	2.4x
Pin	100%	4.7x
Pin	10%	5.5x
Pin	1%	5.3
Dyninst	100%	897x

TABLE 3.4: *A comparison of memory address trace collection only for different binary instrumentation tools.*

code be replaced or removed in such a way that no thread is ever executing it. Also, this on/off mechanism somehow suggests that sampling is enabled/disabled at a global level, that is for either all threads or no thread. This contrasts with the multi-mode approach adopted by Hijacker, which at least in principle allows each thread (even those sharing the same code) to execute in a different instrumentation mode.

CHAPTER 4

Virtual Page Tracing

It's incredibly simple, like all difficult things.
— PKNA #14 - Carpe diem (1998)

Instrumenting memory accesses, as we learned in previous chapters, is a tricky task that can seriously hamper the profiled applications as soon as it is launched. Fortunately, sampling and buffering can alleviate this problem to some extent. The purpose of this thesis, though, is not that of tracing fine-grained accesses to memory but, rather, that of detecting accesses to virtual pages. There are many reasons for which such a goal may actually prove useful for analysis tools, some of which are explained below.

- Operating systems like Linux are oblivious to user-level fine-grained memory accesses. A process' address space is seen as a set of virtual pages, indexed by a convenient hierarchical structure which works on page addresses rather than full addresses. For this reason, many of the system calls and services which can be used to manipulate the address space of a process (e.g., `movepages`) work at the page-level granularity [26] [27]. Hence, it is convenient for some analysis task to adhere to this perspective on virtual memory.

- Checkpointing, intended as the task of saving the state of an object in order to provide a restore mechanism, may too benefit from a coarser-grained view on memory. This is true, in particular, for those applications where the states to be logged are intrinsically huge, or there is a high number of smaller-sized states which must be logged and they all are coalesced at the logical level. For example, an interesting application of virtual page tracing on checkpointing scenarios is due to migration of simulation objects in a PDES platform like ROOTSim [8], together with their states [28]. More details on checkpointing in HPC systems can be found in [29] [9] and [30].

- Virtual pages are the ideal unit of measure to track the working sets of threads in a process, or those of processes in a system. In HPC

scenarios, we can indeed expect applications to have localities that span multiple pages. In this case, keeping track of accesses in terms of fine-grained data structures is unproductive and excessively expensive. The ideal granularity at which the application is issuing accesses is no more that of generic memory addresses. To efficiently trace data accesses at such levels, one better copes with virtual pages.

- Detecting page accesses rather than full accesses can create a new interesting form of sampling based on access granularity. Simply speaking, if two accesses are detected to fall within the same page, it's not necessary to instrument both. Therefore, sampling is driven by the relative address distance between the two accesses under comparison. As a result, one may expect the percentage of final instrumented accesses to be considerably lower than those instrumented for fine-grained access tracing. In practice, this depends on many factors, as I will explain later in this chapter.

My proposal relies on two complementary ideas. First of fall, finding 'hot' or somehow relevant code regions to instrument, while discarding others. Then, instrumenting these regions according to a probabilistic detection of virtual pages. By introducing appropriate parameters, this mechanism can be tuned toward accuracy or toward efficiency, according to our needs. Moreover, compared to classic memory tracing approaches where parameters tend to be somehow arbitrary, the nature of these new parameters makes the above mentioned trade-off much more explicit.

As hinted in previous chapters, these techniques are developed for the Hijacker project using static binary instrumentation, the ELF relocatable file format and x86-64 as the reference architecture The first problem that we must deal with is that of finding relevant code regions to instrument. It is highly theoretical and involves notions and concepts that lay the foundation of compiling theory. Later, the problem of actually instrumenting such regions will be explored from a much more practical perspective, thus involving concrete Linux, ELF and x86-64 assembly concepts.

4.1 Finding relevant code regions

The problem of finding relevant code regions can be seen as the composition of many different sub-problems. The first is deciding the granularity at which code regions are defined. The second is giving a quantitative definition to the concept of relevance. The third and most complex one is performing the right kind of analysis on code regions at the chosen granularity to extract the desired relevance value.

For the purpose of this thesis, code regions are minimal (see chapter 3) and relevance is mainly defined as the number of times that a region participates to a cycle in the program. Many other features can be derived though. For example, the number of memory operations performed within the region may be an indicator of how relevant that region can be for us. Another feature may be the presence of particular kind of operations (e.g. vectorial and floating-point ones).

Among all the possible features, participation to cycles allows to understand at compile-time which kind of regions will have a likelihood to make up for the vast majority of the execution time of the entire application. Such regions are therefore 'relevant' because by focusing on them, while discarding all the remaining ones, we can make a first significant step toward the efficiency goal. After all, a region that never lies on a cyclic path will be ephemeral by definition. As such, the effort put into instrumenting won't be compensated by the pay-off.

The following sections will deal separately with each of the above-mentioned problems, by first devising a way to compute minimal code regions, then proceeding to evaluate the number of different program cycles to which every such region participates to.

4.1.1 Basic blocks and control flow graph

In compiling theory, it is typical to represent the execution flow of a program in terms of a *control flow graph* (CFG) where nodes are minimal code regions and edges represent the existence of a control flow hijacking from the source region to the destination one [31]. Program cycle detection, along with other analysis tasks, can easily be articulated after this structure. Before providing a formal definition of a CFG, let's give a deeper look at the properties that make up its nodes. In the CFG, a node is called a *basic block*, and is defined as follows.

Definition 1. *A basic block is a contiguous sequence of instructions with one entry point at the first instruction and one exit point at the last.*

Informally, a basic block is such that execution can only flow through it from the first instruction all the way to the last. Notice that it doesn't mean that there can only be an incoming edge and an outgoing edge. In fact, in a CFG the in-degree and out-degree of basic blocks is typically larger than one. The meaning of the above sentence, therefore, is only that if we were to split basic blocks into as many nodes as the number of contained instructions, incoming edges would reach the first node resulting from the splitting, while outgoing edges would depart from the second.

By definition, no instruction in a basic block except the last can be an instruction that divert the execution flow. In Assembly, such instructions

are jumps, calls to routines and calls to interrupt or exception handlers (there might be others, depending on the ISA being discussed.) Indeed, was it in the middle of a block, it would violate the property that incoming flow must necessarily go across the entire block to leave it. For similar reasons, the destination of a diverting instruction cannot be in the middle of a block too. Otherwise, flow could enter a block without passing through its first instruction. To sum it up in a few words, if control flow runs into a basic block, it can only enter it from above and leave it from below. There cannot be sideways entrances or departures.

Perhaps surprisingly, the above definition of basic blocks allows multiple implementations of the same concept. Provided that a code region adhering to the definition cannot be made larger without violating the definition itself—as it would contain either a diverting instruction itself or a diverting instruction's destination—they can be made as smaller as desired. In my work, basic blocks are maximal, in the sense that no two directly connected basic blocks can be merged without violating the definition. Observe that this is exactly the same definition that was given in chapter 3 for minimal code regions. Albeit confusing, the adjective 'minimal' there suggests that we reached the definition from larger regions, while the term 'maximal' implies that we approached it from smaller basic blocks.

Now that we've defined basic blocks, it is time to give a more exact definition of a CFG. We've already mentioned that it represents the control flow of the application in terms of the possible trajectories that can be drawn over code.

Definition 2. *A CFG is a graph structure where every possible path represents a legal execution path for the described application.*

Although this definition may result obvious, it hides a subtlety. Suppose that a CFG is created for the entire program where call instructions are included in the diverting instructions. Now, imagine to have a function which is called in different places in the code. Such instruction will too be represented as a graph of basic blocks, perhaps with a single entrance block but more frequently with multiple exit blocks. The question now is: to which node in the CFG should each of such exit blocks point to? If the callee function has multiple callers, one might simply decide to let them link to every possible return block of every possible caller. Unfortunately, by doing so, we are violating the property of CFGs: a function, apart from exceptional cases, never returns to a different caller than the one which actually invoked it.

For this reason, it is natural to define multiple CFG, one for each function. This way, the correctness of the graphs is always preserved—every path in every graph represents a valid program execution path within that

function. To represent the fact that functions may call each other, a higher-level structure is defined called *function call graph* (FCG). At this level of abstraction, a node represents a function, and an directed edge between two nodes encodes the fact that the source function is calling the destination one. In the end, by decoupling the original CFG into as many CFGs as the number of functions, along with a global FCG, control flow analysis becomes much simpler.

4.1.2 Computing basic blocks

There are at least two ways to derive basic blocks for a program function. The first follows a bottom-up approach, while the second a top-bottom one. The *bottom-up* algorithm creates as many basic blocks as the number of instructions in the function, then connects these instruction-level blocks on the basis of which instruction can follow the current one at execution time. By the end of this phase, we have a CFG which must be finalized by merging basic blocks until they become maximal. The merging strategy is actually straightforward: for any pair of connected blocks, try to merge them without violating the definition. This process continues until there isn't any pair which can be merged anymore.

The *top-bottom* algorithm replaces the merging strategy with a splitting one. A big block is first created for the entire function. Instructions in the block are then parsed one-by-one until a diverting instruction is met. At this point, two splitting operations occur respectively on the block that contains the diverting instruction and that which includes the target one. The diverting instruction becomes the last instruction of a newly created block, while the target instructions becomes the first one of another, newly-created block. The process continues until we reach the end of the function.

Let's give a name to the set of all instructions that either divert the flow, or are the target of a diverting instruction—the *boundary set*. Then, the number of merges is roughly equal to the number of instructions that aren't in the boundary set, while the number of splitting operations is equal to the boundary set cardinality. Hence, the choice of one approach over another is naturally driven by an estimation of the ratio of boundary instructions over non-boundary ones. If this ratio is greater than one, then the merging strategy will be faster because it minimizes the number of operations to reach a compact CFG. In the opposite case, the splitting strategy is better for literally the same reason.

On average, user programs—included highly interactive ones,—have a much larger number of non-boundary instructions than boundary ones. For this reason, this thesis sticks to the splitting strategy. The pseudo-code for the splitting algorithm is reported in 4.1, while an non-operational definition

is proposed in provided in [32]. Its running time can be expressed in terms of the number of instructions that make up the function n, the number of diverting instructions d, the number of target instructions t and the costs for routines splitFirst, splitLast and lastly findBlock (whose pseudo-code is not provided as it would be largely implementation-specific):

$$O(n + d \times \mathcal{C}(\text{splitLast}) + t \times \mathcal{C}(\text{splitFirst}) + t \times \mathcal{C}(\text{findBlock}))$$

Algorithm 4.1 Block splitting algorithm

Require: Function $func$
Ensure: Maximal CFG($func$)

```
function splitFirst(blk, instr)
    next(blk) ← ⟨instr, last(blk)⟩
    blk ← ⟨first(blk), prev(instr)⟩
end function

function splitLast(blk, instr)
    next(blk) ← ⟨next(instr), last(blk)⟩
    blk ← ⟨first(blk), instr⟩
end function

currentblk ← ⟨first(func), last(func)⟩
instr ← first(func)

for all instructions instr in func do
    if instr > last(currentblk) then
        currentblk ← next(currentblk)
    end if

    if instr diverts flow then
        splitLast(currentblk, instr)
        for all destinations target of instr do
            targetblk ← findBlock(currentblk, instr)
            splitFirst(targetblk, target)
        end for
    end if
end for
```

4.1.3 Computing program cycles

The task of computing participation to cycles can be easily divided into two different problems:

1. finding program cycles in a given CFG
2. propagating participation to program cycles along function calls in the FCG

Finding program cycles in a CFG

As far the first problem is concerned, it must be noted that is is probably the hardest to solve. Indeed, program cycles and CFG cycles aren't the same thing. More specifically, while it a simple CFG cycle is always program cycle, not all composite CFG cycles are program cycles in general. Recall that a simple graph cycle is minimal, i.e., a cycle whose nodes don't participate to any other cycle. On the other hand, a composite cycle is such that a subset of its nodes belong to a simple cycle. Perhaps more surprisingly, while a CFG cycle is by definition a sequence of nodes, a program cycle can be succinctly expressed as a *loop header* and a set of *back edges*.

Conceptually speaking, a *loop header* is a basic block that must be necessarily visited before entering the next cycle iteration. Notice that it's not mandatory to visit this block to enter the cycle (e.g., `do-while` statements), nor it is mandatory to visit it upon leaving the cycles (e.g., `break`). However, such block must always be visited before getting into the next iteration (even when the `continue` construct is used). As such, a loop header is a perfect candidate to form a 1:1 correspondence with program cycles in the CFG.

The loop header alone allows to establish the presence of a program cycle, but it cannot tell the blocks that make up its body. To accomplish this task, back edges must be identified. A *back edge* is an edge in the CFG whose destination block is always visited before the source block in any depth-first CFG traversal run from the beginning of the CFG. A formalization of these concepts relies on the definition of the *dominance* relationship between blocks in the CFG.

Definition 3. *A block d dominates another block s if and only if d belongs to any path from the entrance block up to s included.*

Definition 4. *An edge from source s to destination d is a back edge if and only if d dominates s.*

Definition 5. *A block h is a loop header if and only if it is reached by a back edge.*

Based on the above, the problem of identifying program cycles in the CFG is equivalent to the problem of identifying back edges (and therefore loop headers) in the same CFG structure. The algorithm is presented in listing 4.2, 4.3 and 4.4, one for each phase. In phase 1 (4.2), an ordinary DFS traversal is performed on the CFG to recognize both back edges and loop headers. The cost of this part is therefore equivalent to the cost of a DFS algorithm.

Algorithm 4.2 Program cycle detection: phase 1

Require: CFG($func$)
Ensure: CFG'($func$) with loop headers and back edges appropriately tagged

```
function DFSVisit(blk, edge)
    if visited(blk) is True then
        if active(blk) is True then
            Tag blk as a loop header
            Tag edge as a back edge
        end if
        return
    end if

    visited(blk) ← True
    active(blk) ← True

    for all destinations dest of blk do
        DFSVisit(dest, (blk, dest))
    end for

    active(blk) ← False
end function

DFSVisit(first(CFG(func)), ⊥)
```

In the second phase (4.3), for all distinct loop headers in the CFG, another DFS traversal in performed in the reverse direction, i.e. from destination to source. Each of such DFS traversals is therefore rooted at a different loop header. It visits blocks backwards, by only taking paths starting with a back edge. They cannot reach any of the dominators of the loop header, because all paths to them begin with a straight edge (i.e., not a back edge). Furthermore, they cannot leave the loop in the forward direction because of the reverse traversal. As such, the only nodes which can actually be visited

are all the nodes that belong the program cycle rooted at the current loop header. There's a problem, though. If this program cycles contains other nested cycles, they will be visited too. This is something that we wish to avoid, since by definition those nodes belong to a different, although nested, program cycle. For this reason, whenever a loop header is encountered while exploring another program cycle, its back edges are excluded from the current traversal. This is perfectly safe, since there's another DFS traversal already scheduled (or even completed) for that very same loop header.

The cost of a single DFS traversal is therefore related to the size of the program cycle being explored, defined as the number of nodes and edges that make up its body, minus the size of any nested program cycle. By iterating on all loop headers, each header is visited at most twice. Once during the DFS traversal rooted at it, once during another DFS traversal involving an immediately upper cycle. The global cost of this phase can be seen as the cost of a single DFS traversal on a graph with all nodes and edges internal to any distinct program cycle (with certain loop headers possibly counted twice). Provided that a single visit to a single block has a cost proportional to its in-degree, we can estimate the cost as a function of the number of cycles c, the number of per-cycle blocks b_c and edges e_c.

$$\mathcal{C}(\mathrm{DFS}(\sum_i b_i, \sum_i e_i)) \leq C \leq \mathcal{C}(\mathrm{DFS}(c + \sum_i b_i, \sum_i e_i))$$

In the end, the algorithm must come up with a method to link blocks in the CFG to the closest (i.e., most nested) program cycle they are into, if any. Even loop headers must be linked this way whenever they are, too, part of another cycle. By doing so, computing the participation of a single block to multiple cycles (from inmost to outermost) is quite easy. The loop membership chain that the algorithm has created simply needs to be traversed starting from the current block and moving, at each step, to its loop header. The cost of this final step is equivalent to the number of blocks in the program, times the longest membership chain, which however cannot exceed c.

Propagating program cycle participation across CFGs

Now that we've computed the degree of participation of cycles for each basic block within a single CFG, it necessary to look at the FCG to see whether there are functions that are invoked from cycle contexts. To do this, functions are parsed one-by-one and the list of caller blocks per each function is examined. By the end of this per-function step, the caller block with the highest participation to cycles propagates its score to all the blocks that make up the current function.

Algorithm 4.3 Program cycle detection: phase 2

Require: CFG'($func$)
Ensure: CFG"($func$) with loop bodies appropriately tagged

```
function DFSReverseVisit(blk, edge, header)
    if edge is not a back edge ∨ visited(blk) is True then
        return
    end if

    visited(blk) ← True
    header(blk) ← header

    if blk is a loop header ∧ blk ≠ header then
        for all sources src of blk such that (src, blk) is not a back edge do
            DFSReverseVisit(src, (src, blk), header)
        end for
    else
        for all sources src of blk do
            DFSReverseVisit(src, (src, blk), header)
        end for
    end if
end function

for all loop headers header do
    DFSReverseVisit(header, ⊥, header)
end for
```

Algorithm 4.4 Program cycle detection: phase 3

Require: CFG"($func$)
Ensure: Basic blocks with cycle feature computed

```
for all blocks blk in CFG(func) do
    header ← header(blk)
    cycles(blk) ← 0
    while ∃ header do
        cycles(blk) ← cycles(blk) + 1
        header ← header(header)
    end while
end for
```

Observe that without this step, we wouldn't be able to capture the fact that certain functions, and therefore all the blocks that constitute them, are actually part of program cycles contained in other functions. This is a pretty common case in user applications, as function-level modularity speed-ups development and aids long-term software maintenance.

4.1.4 Ultimating basic block features

Once all basic blocks have their features computed, it is time to flatten them into single quantities. Absolute score values, also called weights, are obtained by summing up the value of all features computed for the basic block. Once all absolute weights have been computed, the heaviest block has its value used as a reference for computing relative scores. In the end, all blocks in the program will have weights in the $[0, 1]$ range, with 1 referring to the block(s) with the biggest absolute score.

This step will later allow to compare blocks by means of comparing their scores, to eventually decide whether a basic block will be instrumented or not. The acceptance threshold $\mathcal{T}$ will represent the lowest relative score that a block must posses in order to qualify for the actual instrumentation. Observe that low values of $\mathcal{T}$ increase the accuracy of the instrumentation, since a higher fraction of virtual page accesses are intercepted. At the same time, it increases the overall overhead experience at run-time by the instrumented application, because a higher number of blocks are instrumented. High values of $\mathcal{T}$, on the contrary, create lightly instrumented applications to the detriment of tracing accuracy.

Finding the optimal value for $\mathcal{T}$ is a problem that has different solutions for different applications. It depends on the features that are computed per-block, as well as the characteristics of the application itself. In the end, different distributions of block relative scores are produced by varying the application or the application's features. The shape of each distribution in the ends determines which is the actual relationship between $\mathcal{T}$, the accuracy and the overhead, hence the possible trade-offs that can be achieved in the current set-up.

4.2 Detecting virtual pages

The second, big issue to solve concerns the identification of the actual virtual page accesses that occur in the code. The difficulty of this task lies in the fact that we're working on relocatable files, i.e., an intermediate result of the compilation process. As already explained in chapter 2, relocatable files rely on the notion of offsets or displacements from the beginning of sections. The final virtual addresses of code and data are not available at

this stage and will be computed in the final linking phase. Therefore, virtual page addresses too aren't available, since they can be obtained by masking the virtual address relative to a memory access against an appropriate bit pattern.

The absence of virtual (page) addresses is a calculated risk of instrumenting relocatable files. It poses an interesting challenge: devising an appropriate heuristic function to determine whether any two memory accesses will fall within the same virtual page. Therefore, the rest of this chapter is devoted to finding and implementing such heuristics for the sake of accuracy and efficiency goals.

The basic intuition of this thesis is to instrument each basic block by taking into account the partitioned model of relocatable files. Memory accesses are performed from the code section to a data section. A data section, in turn, can be a read-write or read-only section, as well as a section which has initialized or non-initialized (therefore zero'd) data. Furthermore, a data section might be private to a thread or be shared on a process scale. Last but not least, such accesses are encoded through relocation towards symbol, and are filled with several meta-data concerning the positioning of the symbol within this model and its size.

By leveraging on this organization of relocatable files, it is possible to trace a memory access back to the section where it belongs, as well as probabilistically trace it back to the final virtual page that will hold it. Unfortunately, the address of the virtual page being identified is not known in advance, so it must be resolved at run-time by instrumenting the original code. Not only that, but instrumentation must be carried out efficiently, while also taking into account the presence of multiple threads in the application. Starting from this rough idea, I'll further elaborate on each different issue which must be solved in order to turn this into a feasible solution with a concrete implementation.

4.2.1 Tracing back the section

To understand the importance of this issue, it is worth noting that, at least on Linux, there's a clear mapping between the sections that make up a relocatable file and the actual segments in the final executable which get loaded in memory. Virtually all data sections of an ELF file (e.g., `.data`, `.bss`, `.rodata`) have a direct segment equivalent in memory. For this reason, tracing back the sections associated to accesses allows us to establish the segment they will fall into at run-time even without knowing the actual memory addresses.

Now, it must be noted that on x86-64 virtually all memory accesses can be expressed as a combination of four quantities. The *base register b*

specifies the base address from which the address computation starts. The *index register* i is used as an index value whenever b points to an array-like object. The *scale*, too, is used for array accesses to specify the size of a single array entry and it can assume one of the values in $\{1, 2, 4, 8\}$. Together, scale, index and base constitute the so-called *SIB expression* (Scale-Index-Base). To obtain the final memory address, a displacement d is summed to the result. The final formula is the following and is referred to as :

$$d + b + i \times s$$

Since our task is that of determining the segment an access belongs to, we must devise a way to trace back the segment by only looking at the actual expression that makes up the address. The index register and the scale are generally combined into an offset from the base register, therefore they can't give up any useful information. The remaining candidates are the displacement and the base register. The base register may prove useful, provided that we can establish a mapping between the name of the register and the accessed segment. Unfortunately, this is hardly the case for arbitrary accesses since the vast majority of the register in the x86-64 architecture are general-purpose. As such, when used in an address expression, their names have no particular meaning.

The displacement is the only one remained. Whenever an access refers to an object in a data section which is statically defined into the relocatable file, its value is zero but associated with a relocation entry which specifies the name of the symbol being accessed and the section the symbol belongs to. Therefore, it is the perfect candidate for the job. On the contrary, for accesses to dynamically-defined data sections (e.g. the stack and the heap), the displacement is of little purpose and we are left with the base register and the little to none information that it may leak.

Based on the above, we can distinguish between two basically different kinds of accesses: symbol accesses and flat accesses. *Symbol accesses* are associated with a relocatable entry, which describes the section containing the symbol together with the symbol itself. *Flat accesses* are what remains when the access is directed toward a dynamically-defined segment, that is when there's no relocation we can look into. As for the latter kind of accesses, we can distinguish between stack accesses and heap ones. According to the System V Linux ABI for the x86-64 architecture [33], for accesses that displace from the current stack frame pointer the base register is defined to be either `rbp` or `rsp`. Therefore, knowing that the base register is the base pointer (or the stack pointer) is sufficient to establish that we are dealing with the stack segment. In contrast, when the base register is a general-purpose register, the address is very likely to fall into a dynamically-allocated general-purpose memory region. Therefore, when-

ever the base register is not a stack register, the accessed is considered to be toward the heap.

4.2.2 Tracing back the virtual page

Once we can relate two accesses at the segment-level, establishing whether two accesses are likely to fall into the same virtual page is just a matter of difference between displacements, both in the symbols being accessed from the beginning of that segment (for symbol accesses) and in the actual displacements contained in the instruction (for flat accesses). Put differently, even if we cannot know precisely if two addresses will fall into the same virtual page, we can make an educated guess.

The idea is as follows: given any two accesses A_x and A_y in the same segment, we instrument the instruction $I(A_y)$, provided that $I(A_x)$ was instrumented, if and only if the difference between their displacements fall outside of a $\mathcal{H}$-byte sized area called *estimated page size*. If A_y gets instrumented, it becomes part of the *instrumentation state* and its value is used to perform comparisons against newly encountered accesses. This is also the case whenever no previous access A_x is found in the instrumentation state, meaning that A_y represents the first observed access to the segment. Otherwise, the value of A_y is simply discarded, meaning that A_x is already believed to fall within the estimated virtual page.

The actual heuristic function is therefore:

$$d(A_x, A_y) = A_y - A_x \geq \mathcal{H}$$

Notice that the higher the value of $d(A_x, A_y)$, the higher the probability that the two address will actually be in two different consecutive estimated pages. If we call N the size of an actual virtual page handled by the operating system, then:

- when $d(A_x, A_y) \leq N$, there are only $d(A_x, A_y)$ different ways for a virtual page not to contain both the instructions, hence the probability that A_x and A_y will be in two different pages is $\frac{d(A_x, A_y)}{N}$;
- when $d(A_x, A_y) > N$, the two addresses will always fall into different virtual pages.

Rather than working directly on the distance, we can operate on the value $\mathcal{H}$, which must not necessarily coincide with N. Since we claim that the instrumentation rule does not apply when $d(A_x, A_y) < \mathcal{H}$, then the probability that A_x and A_y will be in two different virtual pages is:

$$\mathcal{P}(\text{False negative}) < \frac{\mathcal{H}}{N}$$

Similarly, we instrument A_y if and only if $d(A_x, A_y) \geq \mathcal{H}$, hence the probability that A_x and A_y will be in the same page is:

$$\mathcal{P}(\text{False positive}) \leq 1 - \frac{\mathcal{H}}{N}$$

For this reason, tuning the value of $\mathcal{H}$, therefore the size of an estimated virtual page, will allow controlling the accuracy vs. efficiency trade-off of instrumentation. Lower values for $\mathcal{H}$ decrease the likelihood of false negatives but loosen the upper bound on the likelihood of false positives. At the same time, higher values for $\mathcal{H}$ decrease the likelihood of false positives but loosen the upper bound on the likelihood of false negatives. In either case we are improving the quality of an upper bound while decreasing the precision of the other. The latter effect doesn't necessarily mean that we're increasing the likelihood of bad events, but only that we're not able anymore to provide a strict confidence bound on their occurrence.

Before considering the issue solved, we must give an answer to two complementary questions. The first question is: how often is the instrumentation state discarded? Answering this question is important because once an instrumentation state is cleared, the heuristic function must re-create its history of already-instrumented accesses. It will instrument again all the first accesses to distinct estimated virtual pages, even when such pages were already captured in a previous instrumentation state. For the purpose of my thesis, the idea is to reset the instrumentation state whenever a new basic block is entered. The choice to work at the basic-block granularity is driven by the fact that a same basic block can be involved into different execution flow trajectories (as explained in section 4.1.1), therefore enabling an inter-block reasoning is an extremely difficult task that is left for future work.

The second question is: how can the distance function $d(A_x, A_y)$ be actually implemented on x86-64 when dealing with ELF relocatable files? As already discussed in the previous section, we distinguish between symbol accesses and flat accesses. Once again, symbol accesses are extremely easy to deal with because we can always use the information contained in a relocation entry to reach a symbol's meta-data. Such meta-data contain, beside a reference to the section which contains the symbol, the displacement from the beginning of the section at which the symbol can be found. Therefore, the distance function is simply the difference between the displacements.

Notice that a more accurate technique would try to determine not only the addresses at which the symbol begins in the parent section, but also the actual displacement within the symbol at which the access is performed. Put differently, comparing displacements from the beginning of the *section* is a thing, but a 100% accurate tracing technique would also compare displacements from the beginning of the *symbol*. Unfortunately, computing these

'second-order displacements' is not possible by only looking at the address expression, since they typically hide behind the index register and the scale value. This is the first approximation that comes from using relocatable files and compile-time instrumentation.

Stack frame accesses are equally simple to deal with, since by convention `rbp` and `rsp` are used to maintain respectively the frame pointer and the stack pointer. While the value of `rsp` changes whenever space for a local variable or a function parameter is allocated on the stack, the value of `rbp` typically remains stable along the whole function. Therefore, for stack accesses that use `rbp` as the base pointer, the distance function is again the difference between the displacements.

Unfortunately, the previously-mentioned System V ABI allows an optimized execution mode where `rbp` is released from its purpose of maintaining the stack frame pointer and turns into a general-purpose register. In this case, all stack accesses are computed relative to the stack pointer `rsp`, which as already explained is subject to frequent changes within a function. The approximation in this case comes from the fact that all `rbp`-accesses are treated as if they all fall within the same virtual page, therefore no distance function is defined. While this is certainly not true in general, the probability of having a false negative depends on the probability of having a stack frame which is larger than the size of an operating system's virtual page, which is typically very low.

The second kind of flat accesses, those directed to the heap, are the most difficult to cope with because the value of the registers that make up the final address is not known until run-time. Moreover, since the heap is a dynamic memory region, it has no associated static information in the relocatable file and can grow and shrink freely. Data-flow analysis technique, in principle, allows to simulate the execution of the program by computing register values or at least tracking changes to register contents. However, they are complex to implement and are of little use whenever register values depend on an external (e.g., user input) or non-deterministic (e.g., random) source.

In the end, in absence of any reliable DFA tool to estimate register contents, we treat each heap access having the same base register as an access to the same virtual page. That is, similarly to `rbp`-based accesses, we again assume there's no distance function to compute. At the same time, two heap accesses are believed to fall within different pages whenever the base register is different. Simply stated, the name of the base register becomes the only discriminant for the heuristic function.

Clearly, this choice has severe limitations. For example, it cannot capture accesses to two very far heap regions which are encoded with the same base register. However, while this scenario is certainly possible, it is my

belief that it will occur as more rarely as the granularity at which the instrumentation state is maintained is reduced. If the instrumentation state is maintained only within a single basic block (as it is in my work), then wen can expect the probability of encountering two very different heap accesses with the same base register encoding to be quite low.

A naive justification to this line of reasoning is that in typical user-level applications, accessing two highly distant entries in the same heap object within the same basic block can be quite rare. Note that 'highly distant' here refers to having two addresses whose distance is comparable to the size of an estimate virtual page, therefore the smaller the size of $\mathcal{H}$ and the higher the likelihood. However, if $\mathcal{H}$ and N have the same order of magnitude, the rareness of the event is preserved. This is due to the fact that heap object are rarely as large or larger than N, but even when they are, most of the time they are accessed once within the same basic block—for instance, once within the same loop iteration. Because of this, when the same base register is used to access two different addresses within the same heap object, then it happens either in a different visit to the same basic block (temporal locality), or in a near basic block in the CFG (spatial locality).

4.2.3 Resolving the virtual page address

For accesses tagged as 'to be instrumented', it is necessary to keep track of the register values that make up the effective address of the virtual page, so that it can be eventually resolved by the application itself at run-time. Since the value of registers is not known at instrumentation time, it is reasonable to resolve virtual page addresses upon encountering the memory access itself. Once this effective address is computed, obtaining the page address is just a matter of masking it against the bit pattern `0xfffffffff000`—in standard x86-64 mode, virtual pages are indeed 4KB large and virtual addresses are 48-bit long.

4.2.4 Devising an efficient instrumentation strategy

So far we know that when an access representing a newly encountered virtual page is instrumented, the instrumentation code needed to resolve its address must be placed synchronously to the access itself. However, we haven't decided yet when to pass this address to the user-defined function that will consume it. In this case, it is appropriate to consider synchronous vs. asynchronous schemes. The former calls the user-defined function upon encountering the access, in the same way as the virtual page address is resolved synchronously. The latter stores the computed address into a buffer

in the instrumented application's address space that is populated at run-time and then flushed at the desired point in execution. As explained in chapter 3, research on memory tracing suggests us that buffering is a nice solution to reduce the instrumentation overhead.

In my thesis, buffering is employed at the function-level. Observe that this temporary structure is not populated within the instrumentation tool, but rather by the instrumented application. Therefore, it doesn't exist at instrumentation time, bur rather in the application's final address space. During instrumentation, the function is scanned for memory operations and the heuristic functions decides which accesses to instrument. Not only that, but the heuristics is also in charge of assigning a unique identifier to every such access, to be used as an index within the buffer. By the end of this parsing, all accesses to be instrumented have been collected, therefore we have established a late 1:1 mapping of accesses with the estimated virtual pages that have been detected. Not only that, but the number of accesses performed to the same estimated virtual page within the same basic block is also computed.

As a result of this pass, a data structure in the instrumentation tool is maintained which stores all basic block instrumentation states that will be encountered while traversing the function. Each instrumentation state is basically equivalent to a list of 'to be instrumented' accesses within the related basic block. Each access is an entry that holds the position that the resolved address will have in the application's address space buffer. It also contains the position in the application's code in which instrumentation has to be carried out. The data structure is later read on an entry-per-entry basis to write the actual instrumentation code into the application, at the previously stored positions. This code resolves the address of the virtual page and then stores it into the application buffer, at the position which was uniquely determined in the first pass.

To refer to this buffer inside the instrumentation code injected into the application, it is necessary to first create a new symbol in the appropriate data section—`.bss` being the best candidate, as the buffer can be considered empty at start-up. Then, we can access to such symbol simply by linking the appropriate access instruction to a new relocation entry that references the symbol itself. Once again, relocatable files make the actual implementation logic amazingly simple.

Now that we have buffer transparently populated by the application at run-time, we must decide the nature and number of the so-called *flush-points*, i.e. points in the execution of the application where the buffer of traced virtual page accesses are consumed by the user-specified function. In my work, these points are (a) before the invocation of a local function and (b) before the end of the current function. These two kind of flush-

points are sufficient to provide an asynchronous flushing mechanism which is both sufficiently frequent—at least for user-level applications with function-level modularity—and not too intrusive—for the sake of efficiency. The only detrimental effect of buffering (and invocation of user-defined profiling functions) is that caches will be polluted with cache-lines resulting from populating the application buffer. This effect, somehow unavoidable, is better described in [13] and [23].

4.2.5 Devising a solid threading model

While dealing with multi-threaded applications, it will be necessary to actually have a separate application buffer for each different thread in the final application layout, similarly to what has been done in [25]. No synchronization should be ever required inside the user-defined routine, as that would slow down the application excessively, thus sacrificing the effort put into devising efficient instrumentation techniques. In practice, rather than working with a single buffer as we assumed in the previous section for the sake of simplicity, we actually need a way to maintain separate disjoint buffers.

The threading model adopted for my work is based on the exploitation of a low-level handling of thread-private memory called *thread-local storage* (TLS). This model, supported by the majority of architectures and operating systems, allows to define particular data sections in relocatable files (as well as data segments in executable ones) that are transparently handled by the operating system and the standard library to provide each thread in the application with a partitioned memory area. Symbols that fall in this area are defined similarly to symbols in other data sections, and the same holds true for relocations.

For ELF relocatable files, the Linux operating system and the x86-64 architecture, the TLS is found within two data sections: `.tbss` and `.tdata`. Their functioning is quite similar to that of their process-level equivalents, `.bss` and `.data`. They only difference is that whenever a thread access a symbol within the process-level structures, there's a single instance of that symbol, therefore the symbol is shared by all threads. On the contrary, an access to a TLS symbol is private, meaning that there are as many symbol instances as the number of currently running threads.

Compared to the implementation logic suggested in the previous section, the only difference is in the way the application buffer must be referred. Again, a symbol must be defined—whose parent section, this time, is a TLS data section such as `.tbss`. Then, a relocation entry must be created to access the symbol through a memory instruction. To tell the linker that the access must be actually performed relative to a thread-private memory area,

the x86-64 address expression is augmented with a new quantity. For TLS accesses, this quantity is a *segment register*, namely `fs`, which by convention is always guaranteed to point to the beginning of this thread-private area. These minimal changes are sufficient to implement a robust threading model which is actually handled in its entirety by the linker and the run-time environment.

4.2.6 Final instrumentation code

The concrete machine-level instrumentation code that is injected inside the output relocatable file can be considered as the iterative application of two different x86-64 machine-level code snippets: the first is devoted to storing the address of an instrumented virtual page access, together with the access count detected for that page within the same basic block. The second snippet is executed upon encountering a flush-point to give control to the external analysis routine.

The actual instrumentation snippet which is responsible for buffering accesses is reported in the code listing 4.2.6. It is located in the instrumented program rightly before the access which is being instrumented, so that the values of the registers plus the scale that constitutes the SIB expression—below referred to as `rbase`, `rindex` and `rscale`,—are already populated with the right values. Observe how the expression that computes the address looks similar to the expression that loads the value found at that same address. It is not by chance that both the instructions support SIB expressions, and it is also not by chance that this x86-(64) ISA feature is being exploited in the snippet.

Once the address has been computed, it must be stored into the dedicated TLS buffer by relying on the `fs` segment register. To displace to the current position within the thread-local storage, a TLS-specific ELF relocation must be created to the TLS symbol representing the buffer. The `index` at which the entry for the detected virtual page is found is again determined in a previous pass and can therefore be hardwired into the generated machine code. As for `BUFFER_ENTRY_SIZE`, it is a constant value which is too copied 'as is' inside the instruction payload. Then, the access `counter` for this page, which was computed previously along with the index, is stored in the second half of the buffer entry through an identical relocation mechanism.

```
1  push %rsi
2
3  # Load the memory address expression into a register
4  lea 0x0(rbase, rindex, scale), %rsi       #
     Possible relocation
5
6  # Derive a virtual page address by masking out the
     12 least-significative bits
```

```
7   shr %rsi, $12
8   shl %rsi, $12
9
10  # Store the virtual page address into the TLS
      buffer at byte position:
11  # index * BUFFER_ENTRY_SIZE
12  mov %rsi, %fs:0x0       # R_X86_64_TPOFF32
      relocation
13
14  # Load the access counter into a register
15  movq counter, %rsi
16
17  # Store the access counter into the TLS buffer at
      byte position:
18  # index * BUFFER_ENTRY_SIZE + BUFFER_ENTRY_SIZE/2
19  add %rsi, %fs:0x0       # R_X86_64_TPOFF32
      relocation
20
21  pop %rsi
22
23  # Instrumented instruction
24  mov 0x0(rbase, rindex, scale), reg      # Possible
      relocation
```

The code snippet responsible for 'flushing' the buffered data to the external analysis routine looks as in 4.2.6. The most important task in this snippet is represented by saving and restoring the processor state as seen by the application prior to invoking the analysis function. In the System V x86-64 Linux ABI document, registers are split between those that must be saved by the caller (*caller-save registers*) and those that must be saved by the callee (*callee-saved registers*). It is important to respect the ABI because any Linux compiler which produces the machine-code level for the analysis routine expects those conventions to be honored. Moreover, since the external routine and the code in the caller context are compiled separately by two different compiler or two different compiler instance, there's no room for cross-function optimizations. This save/restore mechanism constitutes the major source of overhead at run-time, as better explained in 5.

The user-defined analysis routine receives two parameters: the first is the base address of the TLS buffer, the second is the number of entries (therefore of virtual pages) that can have been detected to be touched by the current function overall. It is typical for a function to only touch a subset of pages in each basic block. For this reason, when the analysis function is called, it is important to check whether the virtual page address located at an entry has actually been accesses since the last flush-point. Notice that the access counts for the active entries in the buffer are not automatically reset to zero by the instrumentation code. Although this might be seen as a limitation of my approach, it actually allows one to implement an aggregation mechanism where access counts survives flush-points.

```
1   # Save processor context
2   pushf
3   push %rax
4   push %rcx
```

```
5   push %rdx
6   push %rsi
7   push %rdi
8   push %r8
9   push %r9
10  push %r10
11  push %r11
12  ...
13
14  # Load the TLS base address into a register
15  mov %fs:0x0, %rdi
16
17  # Displace from the TLS base address to the address
      of the TLS buffer
18  lea 0x0(%rdi), %rdi       # R_X86_64_TPOFF32
      relocation
19
20  # Store the total number of pages touched by the
      function into a register
21  mov total, %rsi
22
23  # Call the user-defined routine
24  call routine              # R_X86_64_PC32 relocation
25
26  # Restore processor context
27  ...
28  pop %r11
29  pop %r10
30  pop %r9
31  pop %r8
32  pop %rdi
33  pop %rsi
34  pop %rdx
35  pop %rcx
36  pop %rax
37  popf
38
39  # Flush-point
40  # e.g. call isntruction to a local function
41  # or end of function
```

4.3 Contribution to Hijacker

Part of the work put into this thesis involved the actual implementation of the algorithms and procedures explained in this chapter into the Hijacker instrumentation tool. On the one hand, this has been done to advance the development of this software and augment its instrumentation capabilities. On the other hand, a concrete evaluation of the algorithms and technique shown above, like the one carried out in the next chapter, is needed to appreciate the advances in the accuracy vs. efficiency trade-off of access tracing. My contribution to the Hijacker project can be grouped into contributions to the IBR, the front-end and the back-end modules.

4.3.1 Front-end and IBR

The front-end has been augmented with further logic that enables Hijacker to build a CFG for each function in the module to instrument. To do so, changes have been reflected too into the IBR that is eventually build out of this parsing step.

Branch tables

Beside regular jump and call instructions, Hijacker is now able to tentatively build jump tables and call tables respectively for indirect jump and indirect call instructions. Indirect jumps are usually exploited to implement what in C language results into `switch-case` statements, while indirect calls are used for (arrays of) functions pointers. The procedure that achieves so relies on a quite simple heuristics which, upon encountering a branch instruction of any kind, goes backward in the instruction chain until it is able to resolve both the base address for the branch table and its size. Since such information is usually encoded in two different previous instructions, the heuristics stops as soon as it successfully resolves those pieces of information, or upon exceeding the maximum defined look-behind. As a result of this newly-introduced support for branch tables, the instruction chain has been enriched with additional links between instructions resulting from indirect jumps and calls.

CFG and FCG

After instructions, functions and branch tables have been resolved, Hijacker starts a new procedure which, for each function, builds a CFG according to the splitting strategy and the instruction chains previously built. While it does do, Hijacker also checks for call instructions, either regular or indirect, and create a FCG. In the end, the IBR used by the instrumentation engine contains one additional layer for basic blocks, as well as two additional kind of structures: a CFG for each function and a single FCG. Not only that, but while basic blocks are derived by splitting, a balanced binary tree index is maintained that allows to retrieve a basic block from an instruction in a logarithmic number of steps. The adoption of an index is motivated by the fact that the number of basic blocks in a program may well become too high to bear the cost of a linear look-up procedure, especially if such procedure has to be used often during instrumentation.

4.3.2 Back-end

The main contribution to the back-end is represented by the development of the preset system. This is a new Hijacker component which allows to extend

Hijacker with additional functionalities without affecting the core internals. A *preset* can be seen as a specific-purpose instrumentation rule that tries to solve a particular instrumentation problem (for instance, virtual page access tracing) by relying on the APIs provided by the instrumentation engine. Presets can be declared for usage in the rule file, can be passed an arbitrary number of parameters and only require slight modification to the main code-base to register their presence. Everything else—namely, everything which is strictly related to the logic of that particular preset—is bundled into a different C module. The invocation of a preset occurs by passing to the registration procedure an initialization function and an instrumentation function. The former is executed once upon switching to a new executable version, while the second is executed whenever the preset is invoked into the rule file with a particular set of parameters.

As already anticipated, the entire logic for the virtual page access tracing has been implemented into an official Hijacker preset named `vptracer`. The initialization procedure for this preset simply registers a TLS buffer into the IBR which is unique to the current executable version—beside being unique to a thread—and computes the supported features for each basic block in the program. The instrumentation procedure accepts three parameters: `threshold` is an implementation of the threshold value $\mathcal{T}$; `sizeexp` defines a base-two exponent that is used to compute the value of $\mathcal{H}$; `usestack` is a boolean parameter which, when false, tells the preset not to instrument stack accesses—since they tend to pollute the buffer with uninteresting accesses. An example usage for these parameters can be found in chapter 5.

CHAPTER 5

Experimental Assessment

"Forty-two," said Deep Thought, with infinite majesty and calm.
— Douglas Adams, The Hitchhiker's Guide to the Galaxy (1979)

In this chapter, I provide an experimental evaluation of the virtual page tracing technique discussed in my thesis on an application that simulates the dynamics of a GSM-based Personal Communication System (PCS), shown in picture 5.1. Specifically, I intend to assess the goodness of my instrumentation approach both in terms of run-time overhead and with respect to accuracy. The simulation is run according to the principles of Parallel Discrete Event Simulation (PDES) platforms [34], that is by scheduling *simulation events* which correspond to the events that would occur in the physical system under exam. The generation and consumption of these events is actuated by a series of *logical processes* (LP), each of which manipulates a private simulation state. The union of all the private simulation states, together with any state which is globally shared, constitutes the *application state*. The latter represents the logical equivalent of the physical state of the real system, included those parts which cannot be observed in output or stimulated by input.

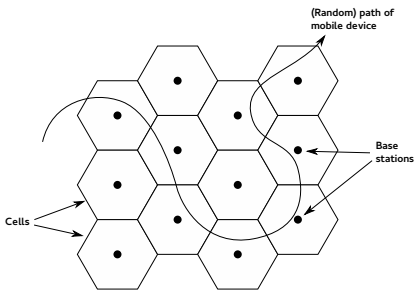

Figure 5.1: *GSM area network example*

In this model, there's a LP for each distinct wireless cell covering a square region. Cells are modeled as hexagons and induce a natural partitioned view of the state of the entire region. Each LP can process or schedule one of

the following event types, corresponding to the event that would occur in an actual wireless cell:

- *Start Call* and *End Call* simulate respectively the beginning and the termination of a call at the current cell;
- *Hand-off Leave* and *Hand-off Receive* simulate the departure and arrival of a pre-existing call from/to the current cell.

The simulation of power regulation and interference/fading phenomena is driven by the results in [35]. Upon the beginning of a new call or the occurrence of an hand-off, the current cell achieves power regulation by looking at the currently allocated power for standing calls into a list of power-management records. Then, the minimum transmission power required to setup the current call is computed, so as to achieve the desired Signal-to-Interference Ratio (SIR). The simulation is highly configurable in terms of simulation parameters:

- S is the size of the square region, expressed as a square number of wireless hexagonal cells;
- N is the number of channels per cells;
- τ_A is the mean inter-arrival time of calls to a wireless cell;
- $\tau_{duration}$ is the expected duration of a call;
- $\tau_{residual}$ is the expected residual residence time of a call into the current cell.

Observe that the arrival distribution of call is considered to follow an exponential distribution for the sake of simplicity, driven by τ_A. The same holds true for $\tau_{duration}$ and $\tau_{residual}$, which together with τ_A make the statistical model quite simple to simulate. The utilization factor ρ of a single channel can therefore be expressed as $\tau_{duration}/(\tau_A \cdot N)$. The impact of ρ on the simulation is significant, as the higher this value and the higher the extent to which channels are busy. This is turn affects the kind of computation performed by a single thread to simulate a wireless cell, as the number of power-management records for currently handled calls increases, thus affecting the execution time and occupied memory to dispose of a single event.

The actual structure used in the simulation platform to schedule and fetch the next event to be processed is the so-called *calendar queue* [36]. As the name suggests, it basically maintains a list of events ordered by timestamps. The event with the last time-stamp in the list is the first to be

processed, hence the first which gets extracted from the list when the next event to dispatch is required by the simulation platform. The peculiar aspect of the calendar queue is that it introduces an additional dimensionality to the classic queue implementation, so it can actually be seen as an array of queues. Taking inspiration from real solar calendars, event timestamps are categorized into days and years. The number of array entries is equal to the number of days that make a year in the current instantiation of the calendar queue. Hence, all the events that are defined to occur within the same day are placed into the same queue.

Inserting an event in the calendar queue entails computing its year and day, so as to locate its queue in the array. Once the correct queue is found, insertion in it is performed in an ordered manner on the basis of the actual event time-stamp, from most recent to furthest in time. It might be possible that two events scheduled for different years share the same day. In this case, they will fall into the same day queue. Extraction of the next event from the calendar queue is performed by querying the time-stamp of the first event in each day, starting from the current day and the current year. These two values constitute a persistent calendar queue state and respectively store the day and year of the last previously-dispatched event. If the first event found at the queue for the current day belongs to the current year, the event is returned to the platform. Otherwise, if the event doesn't belong to the current year, the next day is examined. Upon successive queries on the calendar queue, if there are no more events to be scheduled for the current year—i.e., from the current day to the last day of the current year,—the current year is incremented and a new search begins. As a result, insertion is an $O(d)$ operation where d is the size of a day queue, while extraction is a $O(y)$ operation which y being the size of a year. In practice, the geometry of the calendar queue is optimized so as to make sure that there's always an event to schedule in the current day, therefore the average cost for both extraction and insertion is $O(1)$.

To experiment with the threading model discussed in chapter 4, a multi-threaded simulation is considered. However, contrarily to the generic case of PDES platforms, the implementation that fuels the PCS simulation supports multi-threads but execute them in a sequential fashion. The reasons that led to this design for the testing framework will be clear in the next section. In my experiments, the number of LP that make up the simulation is kept fixed at 1024. Therefore, the square region size S is never changed. At the same time, $\tau_{duration}$ is kept to 120 seconds and $\tau_{residual}$ at 300. The inter-arrival time, on the contrary, is changed in such a way as to produce three different per-cell average utilization factor, namely 25%, 50% and 75%. By examining these three configurations, we have the opportunity to ascertain the load to which each LP is subject to and the actual memory

access pattern that is exhibited.

Lastly, another free parameter of my experiments is the total number of simulation events that a single thread is allowed to process before terminating. This quantity, called N, is an important parameter for our experiments, given that larger values of N lead the application to run less or more ahead of the actual termination condition that have we have selected for the simulation model. This condition has been set to 500 completed calls per cell. Therefore, larger values of N can lead our experimentation to observe what happens when running that simulation application ahead of the phase where the reference termination condition is expected to be matched.

5.1 Worst-case execution time evaluation

The first experiment is aimed at assessing the overhead of the instrumentation approach in a worst-case scenario. The virtual page tracing preset has been run using 1KB as the value of the estimated virtual page size $\mathcal{H}$ and 0.0 as the value of the instrumentation threshold $\mathcal{T}$. This creates a scenario where all blocks are instrumented and the likelihood of a false negative is below 25% (see section 4.2.2). Stack accesses are filtered out because they can be easily caught with dedicated and much more lightweight instrumentation rules that live outside of the scope of this preset. Observe that only the PCS application is instrumented, therefore no instrumentation is performed at the platform level.

In total, this experiment involves 12 tests, split into three different configurations for the utilization factor ρ and four different values for N. The former is allowed to vary in the set $\{25\%, 50\%, 75\%\}$, as explained before, while the latter changes accordingly to the values 10, 100, 1000 and 10000. To estimate the overhead introduced by instrumentation, the preset is instructed so as to invoke an analysis function which does nothing. This way, the intrinsic instrumentation cost of my solution is isolated from that obtained from invoking an actual function which arbitrary contents.

To give a more fair perspective on the slowdown, an adversary tracer is implemented in user-space through the interception of `SIGSEGV` POSIX signals. This tracer is instructed to protect `heap`, `data` and `bss` regions via the `mprotect` system call, so that whenever an access is performed to one of the above-mentioned regions a segmentation fault signal is raised. The newly-installed handler intercepts these signals and does nothing, just like the previous analysis routine, except for removing the protection on the page in which the access occurred. This is done by invoking the `mprotect` system call with reverse flags and prevents the handler from intercepting again the same access.

This reverse protection mechanism inside the handler is needed because

routing signal handling to custom callbacks is an expensive operation which should occur as sparingly as possible. Moreover, to create a worst-case scenario for the instrumentation tracer, it is reasonable to evaluate it against an adversary which operates under the best possible condition. The immediate effect of this choice is that we can only evaluate the number of virtual pages observed by both tracers, while a comparison of virtual pages access counts is not possible. This is not so unfortunate, actually, as it highlights the first strength of the virtual page tracer preset: implementing the same mechanism in user-space by virtue of the above interception of `SIGSEGV` signals is virtually impossible due to the extremely high slowdowns that would be observed.

The high-level functioning of the adversary tracing is as follows. Inside the test framework, the custom handler is installed prior to launching the simulation. Then, between the execution of two subsequent threads, protection of memory regions is enabled, disabled and then enabled again. This is needed because memory accesses belonging to non-simulation variables must not interfere with the tracing, and also because we want to intercept again the accesses performed to the same memory regions by subsequent threads to emulate a threading model. Promoting this enable/disabled logic to a global scale, where the protection is enabled once prior to starting the simulation and disabled once upon its termination, would create a scenario where the adversary cannot distinguish accesses performed by different threads. This, in turn, would make the comparison with the instrumentation tracer completely unfair and pointless. This is also the reason why the simulation is multi-threaded, but run sequentially.

The results for this experiments are depicted in figures 5.2, 5.3 and 5.4. As we can see, the instrumentation tracer maintains a slowdown no higher than 2x over the baseline case with no instrumentation—except for one isolated case. As already explained, this is mostly due to the fact that all the basic blocks in the application are being instrumented, while also trying to increase the likelihood of false positives. This factors being accounted for, the overhead looks quite acceptable. On the other hand, the adversary tracer performs almost equally to the baseline, with the major slowdown coming from N equal to 10 and ρ equal to 25%. That is, when the system is lightly loaded and the number of events executed per-thread is low, the cost of executing the signal handler is poorly absorbed by the simulation. This cost becomes less and less important as N increases , since a higher number of events is executed within the same thread. As a result, the accesses triggered by this additional fraction of simulation events need only be intercepted once, rather than twice from two different thread contexts.

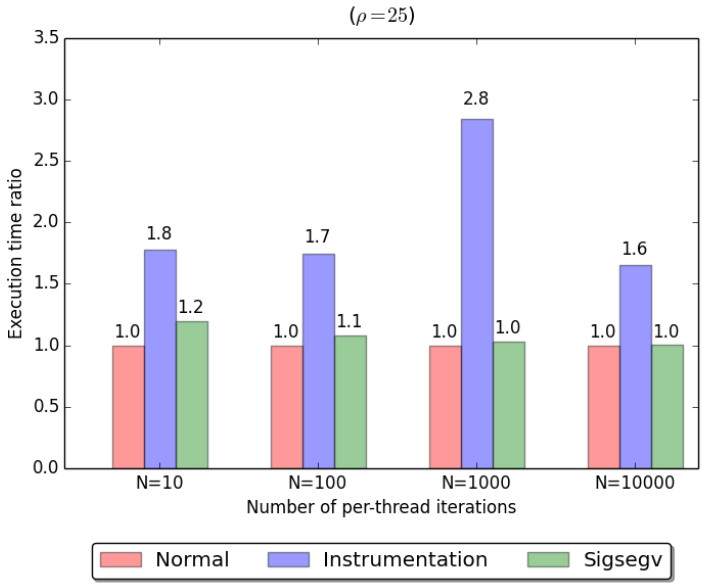

FIGURE 5.2: *Execution time evaluation in the worst-case with* $\rho = 25\%$.

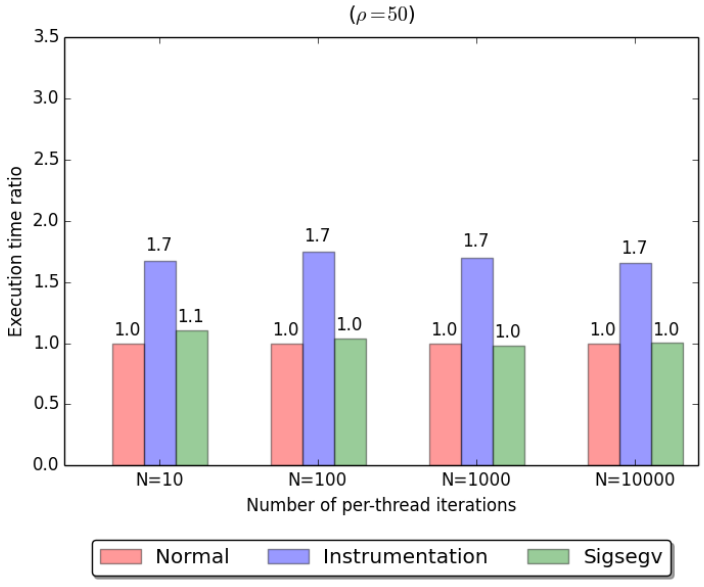

FIGURE 5.3: *Execution time evaluation in the worst-case with* $\rho = 50\%$.

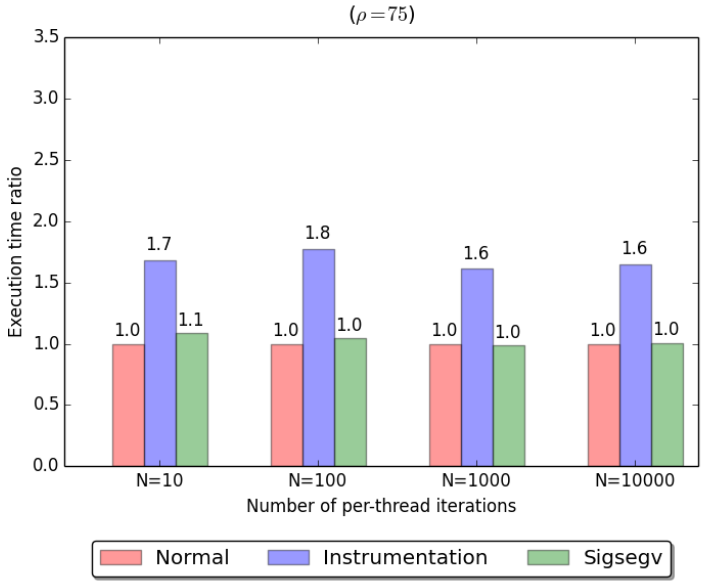

FIGURE 5.4: *Execution time evaluation in the worst-case with* $\rho = 75\%$.

5.2 Best-case accuracy evaluation

The second experiment is devoted to an evaluation of the accuracy exhibited by the instrumentation tracer. The values of $\mathcal{T}$ and $\mathcal{H}$ are the same as the previous experiment, with ρ and N that, too, have the same set of values as before. This is a best case for the virtual page tracing preset because it instruments all basic blocks, and within a single block the probability of false negatives is at most 25%. The accuracy is evaluated relative to the number of different pages recognized by the adversary, since we know in advance that the instrumentation tool cannot detect pages which aren't detected by the user-space tracer.

The outcome of this experiment is shown in figure 5.5. The results match our expectations for 9 cases out of 12. In such cases, we observe that the accuracy of the instrumentation tracer never goes below 96%. In the remaining three cases, which share the assignment $N = 10000$, the accuracy quickly drops to 58%. While this trend may seem unexpected, it has a clear meaning when it comes to the dynamics of the simulation platform. Recall that instrumentation is only performed at the application level, while leaving the simulation environment intact. This means that if the system starts spending more and more time in the platform, the instrumentation tracer won't be able to capture this. Basically, this is exactly what is happening in those three cases.

To explain why such a phenomenon only occurs when $N = 10000$, we

must recall that the termination condition for the simulation is that the number of completed call be at least 500 per logical process. However, evaluating such condition by the end of ever-increasing interval lengths N lead the simulation to start a different phase. In this new simulation phase, is not true anymore that for each event being produced a single event is scheduled. This behavior was indeed part of a transitory, which was what lower values of N actually captured. In the new phase, covered by $N = 10000$, the system stops issuing only *Start Call* events and starts simulating hand-offs, which generate respectively an *Hand-off Leave* and an *Hand-off Receive* at the source cell and the destination one. Given that two events are now being generated at a time, the system starts spending more time in the platform than in the application.

Notice also that when the load is low everything still looks acceptable, since the new phase begins later in the simulation. However, when the load is increased, the hand-offs phase begins earlier. Therefore, most of the simulation time is spent by the system to produce more and more simulation events that will never get executed, because they are scheduled beyond the termination condition. Unfortunately, while the user-space tracer intercepts all memory accesses in the platform relative to such insertions, this part of the simulation is out of the instrumentation boundaries.

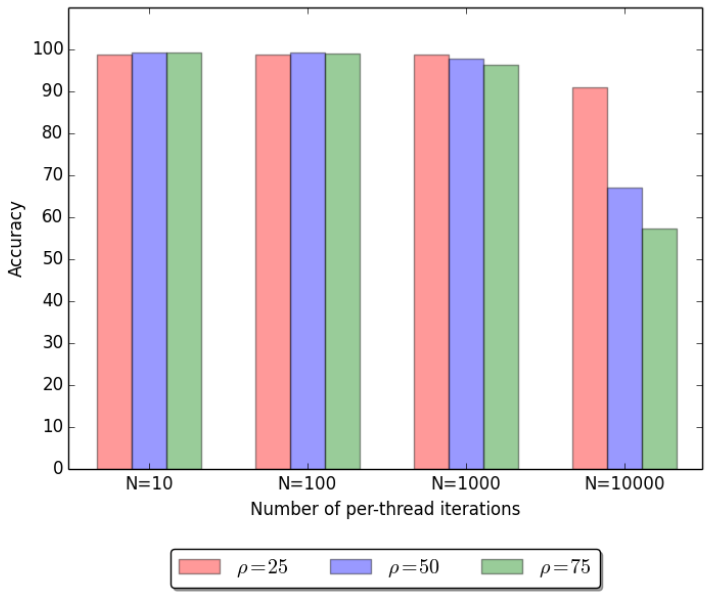

FIGURE 5.5: *Accuracy evaluation in the best-case.*

To better convince ourselves of this fact, it is instructive to look at the number of times the scheduling function and the processing function are actually executed in those three cases. These numbers are produced

via simple access counters which were incremented every time one of the two functions was executed. As we can see from table 5.1, there seems to be a connection between the events scheduled but not processed and the number of pages intercepted by the segmentation fault handler but not instrumented. Once again, the only reasonable explanation for this phenomenon is that before the termination condition is met, the system manages to schedule a residual number of simulation events which clearly increases as the load, too, gets higher.

Scheduling	Processing	%	Case	Accuracy
2'202'116	2'000'000	90%	$\rho = 25\%$	90%
2'607'578	2'160'000	82%	$\rho = 50\%$	67%
2'974'663	2'360'000	79%	$\rho = 75\%$	58%

TABLE 5.1: *Access counts for the scheduling and processing functions when* $N = 10000$

This experiment therefore serves to illustrate the fact that static instrumentation, on the one hand, cannot see instrumentation points which are outside its scope (e.g., the input relocatable file). On the other hand, such a limitation can also be seen as an *opportunity* to instrument different parts of the application of the system with different instrumentation rules, driven by incompatible analysis needs.

5.3 Trade-off assessment

The last experiment is aimed at assessing the kind of accuracy vs. efficiency trade-off that can be achieved via the instrumentation tracer by tuning respectively the values of $\mathcal{H}$ and $\mathcal{T}$. In this experiment, N is fixed to 1000 and ρ to 50%, therefore we are targeting an middle-load case. As opposed to previous cases, $\mathcal{T}$ is allowed to vary among the value $\{0.0, 0.25, 0.50, 0.75, 1.0\}$, while $\mathcal{H}$ can be 1KB, 2KB or 4KB. The configuration with $\mathcal{T} = 0.0$ and $\mathcal{H} = 1$KB should produce the highest accuracy at the expense of the runtime efficiency, while the configuration $\mathcal{T} = 1.0$ and $\mathcal{H} = 4$KB should yield exactly the opposite outcome.

To more evenly distribute block scores, this experiment involves another, previously unseen, block feature. It computes the ratio of encountered memory operations in a single block over the block length. Then, this value is multiplied again by the number of memory operations in a single block, and then divided by the highest number of memory operations seen in the entire function. This feature should give more relevance to blocks with a large

fraction of memory instructions both in a relative (first ratio) and absolute (second ratio) sense.

$$\frac{mem_{blk}}{tot_{blk}} \cdot \frac{mem_{blk}}{max_{blk}(mem_{blk})}$$

The obtained score is then summed to the cycle participation feature to obtain the final absolute block score, prior to converting it to a relative value as section 4.1.4 explains.

As we can see from plots 5.6 to 5.8 and from 5.9 to 5.11, changing the value of $\mathcal{H}$ produces little no incidence on the trade-off. This negative results can be explained by keeping in mind that the probabilistic model discussed in section 4.2.2 is only valid within the boundaries of a single basic block. Stated differently, the instrumentation tracer always tracks an occurrence of the same virtual page if it origins from two different basic blocks. Therefore, at least in the tested PCS simulation, the code almost never happened to access two consecutive virtual pages—which were detected as one single estimate page,—inside the same block.

Speaking about efficiency, we can see that instrumenting less and less blocks had the desired effect of producing more and more lightweight instrumentation. Observe that the values of $\mathcal{H}$ don't represent directly the percentage of blocks being instrumented, but rather the lower instrumentation score that a block must have to become part of the instrumentation process. The actual gain, as already explained in this thesis, depends on the application being tested and the features being provided. For the PCS application and the two block features being computed, namely memory ratio and cycle participation, the actual block score distribution is such that there are virtually no scores ranging in the interval $[0.25, 0.75)$.

As for the accuracy, the obtained results are extremely encouraging. According to the reported plots, even instrumenting only the hottest basic block(s) is sufficient to capture literally all the virtual memory pages ever touched by the PCS simulation. This can be explained by observing that all PCS events can be divided between those that do a lot of computations, namely *Start Call*, and those that almost do nothing, hence the remaining ones. Since the former is actually the one that triggers the actual power regulation, the instrumentation tracer detects the block accessing a single power record as the hottest ever. Once again, hotness is defined in this experiment both in terms of participation to cycles—which are almost non-existent in the PCS case—and the memory ratio feature—which, overall, contributes to the hotness metric to least extent. Therefore, as far as the PCS application is concerned, we get the best achievable trade-off between accuracy and overhead.

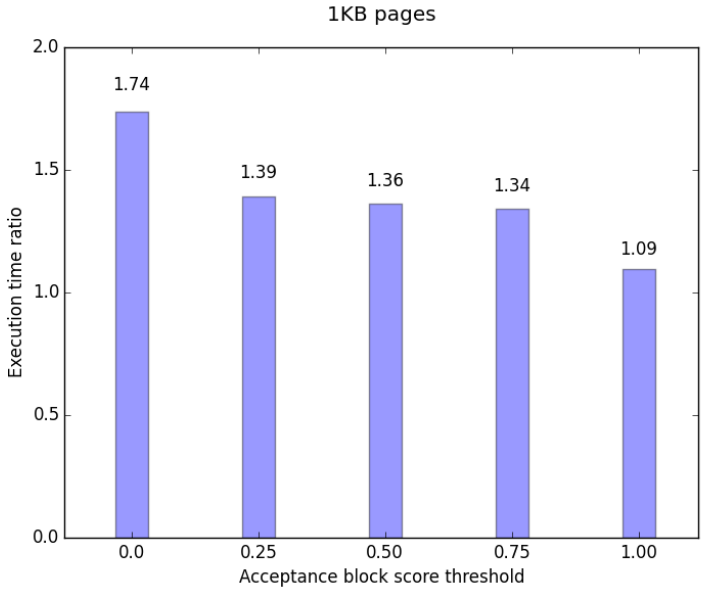

FIGURE 5.6: *Trade-off efficiency evaluation when $\mathcal{H}$ = 1KB.*

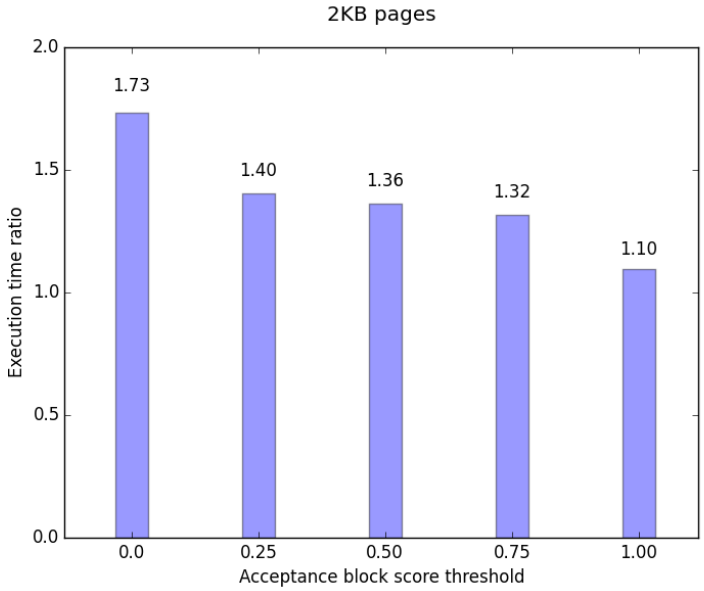

FIGURE 5.7: *Trade-off efficiency evaluation when $\mathcal{H}$ = 2KB.*

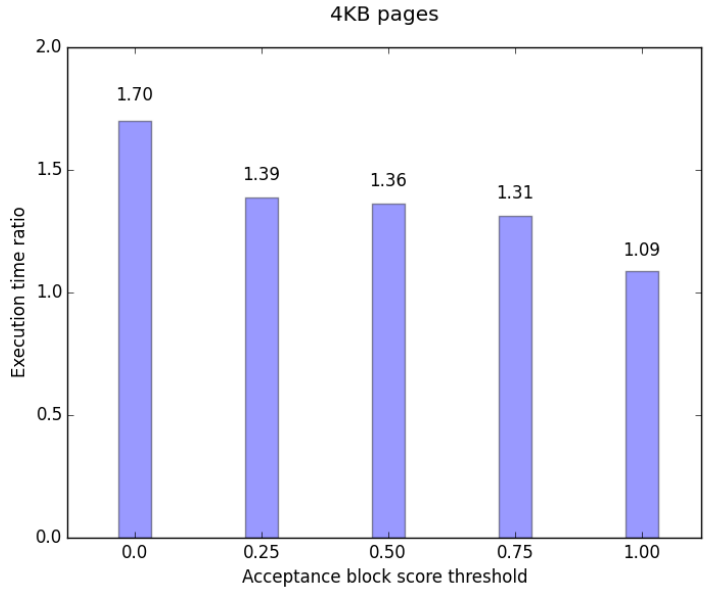

FIGURE 5.8: *Trade-off efficiency evaluation when $\mathcal{H} = 4KB$.*

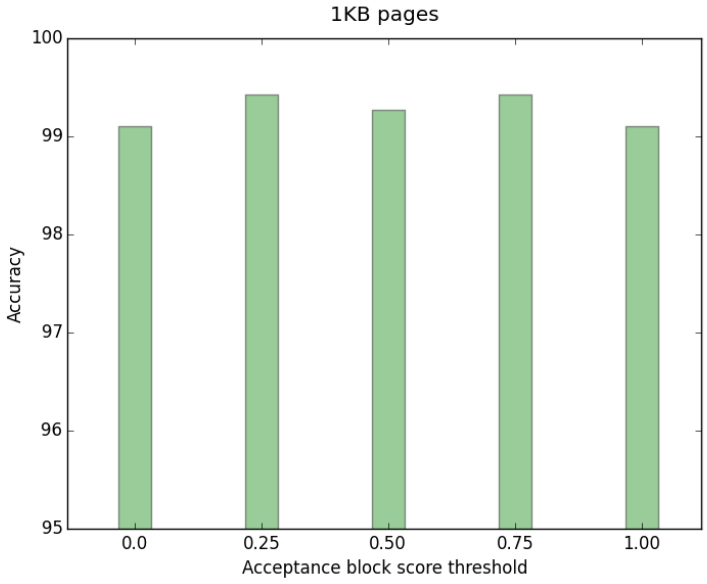

FIGURE 5.9: *Trade-off accuracy evaluation when $\mathcal{H} = 1KB$.*

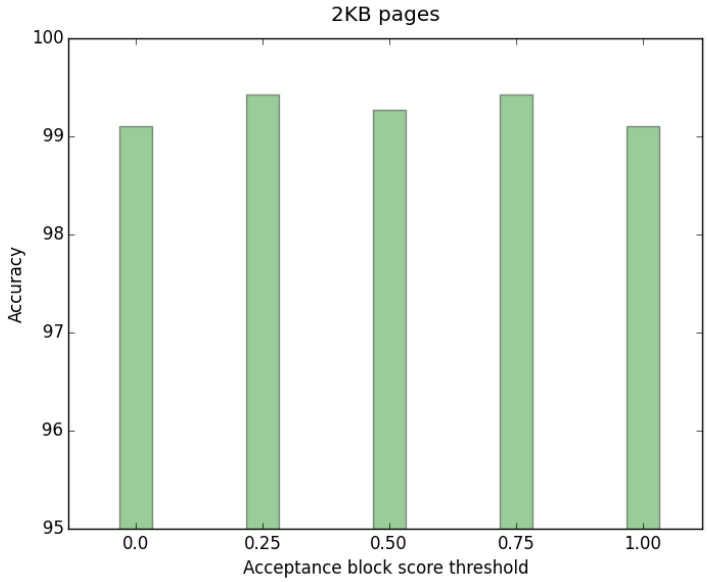

FIGURE 5.10: *Trade-off accuracy evaluation when $\mathcal{H}$ = 2KB.*

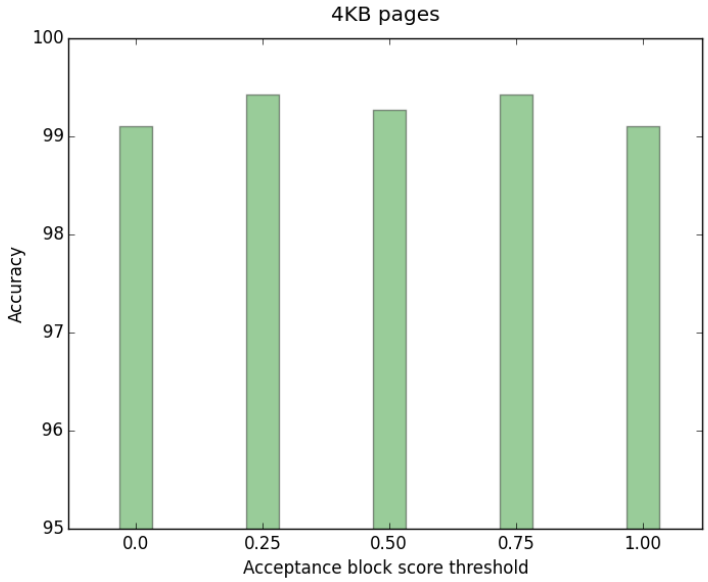

FIGURE 5.11: *Trade-off accuracy evaluation when $\mathcal{H}$ = 4KB.*

CHAPTER 6

Conclusions and Future Work

Roads? Where we're going, we don't need roads!
— Dr. Emmett Brown, Back to the Future (1985)

The work presented in this thesis is an attempt to solve the problem of tracing accesses to virtual pages using static binary instrumentation alone. The idea of using instrumentation to achieve this goal comes from the fact that it is an approach that is completely transparent to the application developer and non-invasive for the system, as opposed to more classic techniques which involve modifications in the operating system, the standard library or even the source code of the application itself. Many static and dynamic instrumentation techniques have been studied in the literature. However, when it comes to lightweight binary instrumentation, static approaches are still the best choice.

Part of the ideas used to optimize the tracing of virtual pages take inspiration from techniques used to trace fine-grained memory access patterns in performance prediction scenarios, such as sampling and buffering. The fact itself that pages are traced, actually, is a form of sampling, since no two accesses that are believed to fall within the same page are both instrumented if coming from the same basic block. Buffering is too employed in my thesis at the function-level, by defining sufficiently-frequent flush-points that, once encountered, cause the an external analysis function to be invoked and the buffer of so-far collected accesses to be consumed.

This thesis also defines a solid and simple threading model based on the reliance on thread-private storage, a mechanism to allocate memory visible to a single thread which is supported by a wide range of architectures and compilers. By virtue of this model, it is possible to support buffering at the thread-level without the need for any synchronization scheme on a shared data structure—which, it must be noted, would hamper the performance of the application in a considerable way. Furthermore, no expensive techniques for retrieving the ID a thread are ever needed, given that the mapping between a thread and its private memory area is transparently handled by the standard library and the operating system.

The presence of two complementary parameters, namely the block-level instrumentation score threshold $\mathcal{T}$ and the size of an estimated virtual page $\mathcal{H}$, allow to control the trade-off between accuracy and instrumentation efficiency according to a probabilistic model which is straightforward to the user. This is better than having arbitrary parameters—such as the interval of instructions between two tracing periods in dynamic sampling,—that give little indication as to how much is lost and how much is gained from using different values. Note, though, that the optimal assignments to these two quantities depends from the application being instrumented and the higher-end goal that one wishes to achieve by collecting virtual page traces.

A series of concrete contributions have been carried out in the form of improvements and additions to the Hijacker instrumentation tool. Specifically, Hijacker's intermediate representation has been extended with an additional basic block layer on which control-flow analysis can be performed. A graph representing calls between functions has too been implemented. Furthermore, Hijacker is now able to infer through a simple heuristic technique the possible targets of indirect jump and call instructions. Last but not least, a preset system has been developed to separate the Hijacker's core logic from the actual instrumentation tasks that can be implemented on top of it.

As a result of the experimental evaluations that have been conducted in the previous chapter on a PDES application, the instrumentation approach is twice as slow on average as the baseline non-instrumented case when full instrumentation is enforced, while also achieving at least 97% accuracy when most of the accesses are performed from within the instrumented application. These results are encouraging and suggest that more time should be invested in trying to reduce the worst-case instrumentation overhead without affecting the best-case accuracy. Moreover, the particular application being tested seems to react extremely positively to perturbations in the value of $\mathcal{H}$ and $\mathcal{T}$, thus suggesting that interesting trade-off balances can indeed be achieved.

A possible application for the work that I've presented in this thesis concerns the estimation of thread working sets. This problem has been explored quite a few times in the literature, but can have many interesting uses. The most relevant is the transparent movement of pages between nodes in a NUMA architecture in order to maximize the fraction of memory accesses that are local to the node where the thread is running. Observe that, compared to classic memory tracing techniques, per-thread working set estimation doesn't need to track individual accesses. Indeed, the granularity at which operating systems act on the address space of a process is the granularity of virtual pages. Therefore, this scenario constitutes the perfect example for a practical exploitation of the virtual page tracing described in this thesis.

6.1 Current limitations and future directions

The results obtained from the experimental evaluations of the techniques illustrated throughout this thesis suggest that the ideas behind them are reasonable and deserve further examination. However, the presented approach is limited by some factors:

- The probabilistic model allows pages to be missed or detected twice, meaning that both false negatives and false positives are possible. The number of such mistakes per block should be quite low, though. If $\mathcal{H}$ = 4KB is chosen, the worst case occurs when considering all expected pages shifted by an amount 4KB-1, therefore at most 1 error can be committed per block. However, in certain situations this could still be too much for the kind of analysis one wishes to perform, especially when high accuracy is needed.

- The virtual page tracer may miss pages whenever sufficient information is lacking in the relocatable file. As an example, the heap is currently not properly handled. Two accesses with the same general-purpose base register are treated as the same segment, but it may happen that between the two accesses, the value of this register has changed significantly. Since we don't perform any kind of data-flow analysis, we are not able to properly capture this situation.

- The instrumentation state is discarded whenever we leave the current basic block to conduct virtual page analysis into another one. While this approach is certainly conservative, it can be improved to a great extent provided that inference on the control flow graph is involved in the process. A simple example of inter-block reasoning comes from allowing each block to see the instrumentation states of all dominator blocks in its proximity. By doing this, a smarter heuristic policy could decide, for example, not to instrument a certain virtual page access whenever it was already instrumented in one of the close block dominators.

- The assembly snippet in charge of invoking the external analysis function currently induces a quite high execution overhead. All caller-save registers must be saved and then restored prior to executing the first instruction from original code. These two expensive steps could be optimized if we only were able to perform liveness analysis on the registers used inside the analysis function, so as to only save those caller-save registers that are actually employed in the callee.

- Currently, Hijacker is not able to properly instrument relocatable files resulting from compiling source programs under a non-zero optimization level. Future work is therefore also oriented toward devising more solid code analysis techniques to detect important program features and disassembly obfuscated code. Specifically, such techniques must be resilient to compiling optimization, meaning that features which are concealed due to the use of advanced optimizations at the machine-code level must nevertheless be detected with the highest possible fidelity.

- Support for multiple object file formats and instruction set architectures should be provided, possibly by relying on external libraries and tools of widespread use in other compiling and development contexts.

To recap, future efforts are devoted to improving the work presented in this thesis from many different points of view: theoretical by devising more accurate heuristics as well as probabilistic models, practical by producing more tailored machine-level optimizations in terms of instrumentation code and instrumentation overhead.

Bibliography

[1] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[2] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[3] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[4] Zoltan Majo and Thomas R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[5] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[6] Henrik Löf and Sverker Holmgren. Affinity-on-next-touch: Increasing the performance of an industrial pde solver on a cc-numa system. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA, 2005. ACM.

[7] Brice Goglin and Nathalie Furmento. Enabling high-performance memory migration for multithreaded applications on linux. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–9, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. The rome optimistic simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 96–98, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[9] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 319–327. IEEE, 2010.

[10] George Nacht and Alan Mink. A hardware instrumentation approach for performance measurement of a shared-memory multiprocessor. In Ramon Puigjaner and Dominique Potier, editors, *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 249–264. Springer US, 1989.

[11] Alessandro Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[13] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183, March 2010.

[14] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4*, 2000.

[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[16] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[17] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[18] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[19] Michael Laurenzano, Beth Simon, Allan Snavely, and Meghan Gunn. Low cost trace-driven memory simulation using simpoint. *SIGARCH Comput. Archit. News*, 33(5):81–86, December 2005.

[20] Laura Carrington, Allan Snavely, Xiaofeng Gao, and Nicole Wolter. A performance prediction framework for scientific applications. In *ICCS Workshop on Performance Modeling and Analysis (PMA03*, pages 926–935, 2003.

[21] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[22] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.

[23] Xiaofeng Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic memory traces. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 46–55, Oct 2005.

[24] Xiaofeng Gao, Beth Simon, and Allan Snavely. Aliter: An asynchronous lightweight instrumentation tool for event recording. *SIGARCH Comput. Archit. News*, 33(5):33–38, December 2005.

[25] M.A. Laurenzano, J. Peraza, L. Carrington, A. Tiwari, W.A. Ward, and R. Campbell. A static binary instrumentation threading model for fast memory trace collection. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 741–745, Nov 2012.

[26] Jeff Bonwick and Sun Microsystems. The slab allocator: An object-caching kernel memory allocator. In *In USENIX Summer*, pages 87–98, 1994.

[27] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[28] Sebastiano Peluso, Diego Didona, and Francesco Quaglia. Supports for transparent object-migration in pdes systems &star. *Journal of Simulation*, 6(4):279–293, 2012.

[29] James S Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical report, Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, 1997.

[30] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems*, PP(0):1, May 2014.

[31] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[32] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, pages 46–55, New York, NY, USA, 2002. ACM.

[33] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface - x86-64 linux. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[34] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

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

[36] R. Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10):1220–1227, October 1988.