8-2010

SimSight: A Virtual Machine Based Dynamic Call Graph Generator

Xueling Chen
University of Nebraska at Lincoln, xueling.chen@gmail.com

SIMSIGHT: A VIRTUAL MACHINE BASED DYNAMIC CALL GRAPH GENERATOR

by

Xueling Chen

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Witawas Srisa-an

Lincoln, Nebraska

August, 2010

SIMSIGHT: A VIRTUAL MACHINE BASED DYNAMIC CALL GRAPH GENERATOR

Xueling Chen, M. S.

University of Nebraska, 2010

Advisor: Witawas Srisa-an

One problem with using component-based software development approach is that once software modules are reused over generations of products, they form legacy structures that can be challenging to understand, making validating these systems difficult. Therefore, tools and methodologies that enable engineers to see interactions of these software modules will enhance their ability to make these software systems more dependable. To address this need, we propose *SimSight*, a framework to capture dynamic call graphs in *Simics*, a widely adopted commercial full-system simulator. Simics is a software system that simulates complete computer systems. Thus, it performs nearly identical tasks to a real system but at a much lower speed while providing greater execution observability. We have implemented SimSight to generate dynamic call graphs of statically and dynamically linked functions in x86/Linux environment. A case study illustrates how we can use SimSight to identify sources of software errors. We then evaluate its performance using 12 integer programs from SPEC CPU2006 benchmark suite.

Acknowledgements

I am deeply grateful to my advisor, Dr. Witawas Srisa-an, for his support, patience and encouragement during my master studies. I could not accomplish my research without his guidance. I am also thankful to him for teaching me academic writing. He is really a nice, humerous and kind person. It is a great honor and pleasure for me to do the research under his supervision. I would like to acknowlege my group member and good friend, Peng Du. Without his encouragement, I could not have joined CSE and met Dr. Srisa-an, my work has benefited from his suggestion and knowledge. My thanks go out to my committee members: Dr. Lisong Xu and Dr. Ying Lu, for their worthful comments and feedbacks on my work.

I want to express my gratitude to Simics engineers on the technical forum, especially Jakob Engblom, *chn* and *simong* (user ID). They have provided such great support each time I encountered an issue about Simics. I also thank Shea Svoboda for helping me administer the Simics license server in CSE.

My gratitude also goes to Dr. Xingwei Wang, my previous advisor in Department of Mechanical Engineering at UNL. With his guidance and kindness I had a very good time. I also want to thank my collegues and friends: Lijun Zhang, Xueming Wu, Du Li, for their encouragement and help.

Finally, I would like to thank my family for their support, especially my husband, Lianan Huang. His support, encouragement and love have made it possible for me to finish my studies. I dedicate this thesis to my grandparents, who passed away in 2008 when I could not go back to my country and have last words with them. I love you all!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's software systems are usually constructed with many software components implemented by different teams of software developers. In building an embedded system, for example, a team of developers may work exclusively on building or updating a runtime system to manage devices (e.g., *Hardware Abstraction Layer* or *HAL*). Another team may work on porting a real-time operating system. Different teams within the company or third-party developers develop applications and libraries. Eventually, these software components interact to perform computing tasks. One major benefit of using this component-based approach is that a team of developers can leverage its expertise to build specific software components that are reusable.

On the other hand, such practice can also lead to some dependability issues especially in embedded systems of which product life-cycles are short but many software components or modules are reused over multiple generations of products. Software engineers have found that once these modules are integrated into generations of systems, they form legacy structures that can be challenging to understand [1]. As these structures evolve, the effects of changes in system components, programming, and configurations can be difficult to predict, making further validation more difficult.

As an example, consider software problems that caused some Toyota Priuses from 2004/2005 to stall or shut down while driving at high speed [1]. To isolate software errors that cause this problem, Toyota engineers need a complete view of interactions and dependencies among these legacy and newly developed modules that make up the drivetrain system. However, they had no tools to obtain such view, so they had to spend a large amount of time to isolate and identify the cause of this problem [1].

1.1 Motivation

Obtaining a complete view of module interactions is challenging in the scenario above because: (i) many of these software modules are legacy, so engineers that fully understand the heuristics and features of these components may no longer be available to provide necessary debugging information; (ii) many of the interactions are implicit, meaning that the module dependencies are not clear; (iii) many modules evolve over time, and thus, part of the code base may be obsolete, making combing through the source code a tedious and cumbersome process; (iv) in systems that comprise of third party or legacy software modules, the source code of these modules may not be readily available; and (v) proprietary systems such as the one controlling the Prius' drivetrain often use in-house software components and development tools instead of off-the-shelf products; hence, finding compatible tools to support testing and debugging can be difficult.

These five characteristics make most existing techniques inadequate to address this module interaction problem because these techniques only work for particular types of modules but do not work across all types of modules. For example, most instrumentation-based approaches only work at source-code level so they cannot capture interactions in modules of which source code cannot be instrumented. Dynamic call graph tools such as *latrace* or instrumentation frameworks such as *DTrace* are operating system dependent so they do not work on embedded devices with no operating systems or systems utilizing unsupported operating systems. According to industry observers, this type of devices accounts for about 60% of all embedded devices or 2.5 billion units shipped in 2006 [2]. Dynamic instrumentation frameworks such as *Pin* can provide such information through binary instrumentation. However, such instrumentation is intrusive since it adds code that may change system states.

Furthermore, most of existing tools do not provide infrastructure for device modeling. Therefore, these tools may not work with executables using low-level hardware-specific code such as device drivers. To ensure dependability of these complex component-based software systems, new tools and methodologies that can provide interaction information are needed [1]. These new methodologies should be developed to meet the following two objectives:

1. must capture interactions among all software modules, regardless of types, without perturbing the system states and
2. must be applicable to embedded systems containing device-specific software components and can be easily adopted by system developers so that they can be used to solve real-world problems.

1.2 Approach

In this thesis, we propose *SimSight*, a framework to generate dynamic call graphs based on virtualization that allows developers to observe the relationships between software modules even if the source files are not available. We explore the use of a commercial full-system simulator, *Simics*, as a way to provide module interactions in the form of dynamic call graphs. We choose Simics because: (i) similar to other full-system simulators, Simics provides functional and behavioral characteristics similar to those of the *target hardware system*, enabling software components to be developed, verified, and tested as if they are executing on the actual systems; (ii) Simics supports several instruction sets including PowerPC, ARM, SPARC, x86, and MIPS [3], making it applicable to various types of computing projects; (iii) through a rich set of Simics APIs, software developers have the ability to non-intrusively observe various system behaviors without ever needing the source code; (iv) because of its powerful device modeling infrastructure, Simics already plays a critical role in hardware/software (HW/SW) co-designs; therefore, adding the capability to observe module interactions to it will enable adoption without requiring much efforts [4]; and (v) licensing of Simics is free for academic institutions, making it a good platform for research.

Our proposed framework is implemented for applications running on x86/Linux environments as a module extension to Simics. We choose Linux for two reasons. First, it provides advanced runtime features such as dynamically linked libraries and multiprogramming that can be a good test for our framework. Furthermore, tools such as *latrace* can be used as an oracle to evaluate the correctness of our framework in generating call graphs.

1.3 Contributions

The presented framework is to take advantage of the non-intrusive probing and execution observability in virtual platforms to improve software quality. Below, we summarize our contributions:

1. We proposed a framework to generate dynamic call graph for both statically and dynamically linked function based on virtualization.

2. We then implemented a prototype of SimSight to support x86/Linux executables based on Simics, a full-system simulator. We detailed the algorithms and the implementation for each of the three components: *SimAnalyzer*, *SimTracer* and *SimParser*. We also illustrate the applicability of SimSight using two scenarios.

3. We evaluated the performance of SimSight and the overhead to capture function call information using 12 integer programs from SPEC CPU2006 benchmark suite.

1.4 Organization

We organized the remainder of this thesis as follows. Chapter 2 describes some relevant backgrounds of the implementation of SimSight. Chapter 3 summarizes existing tools that can generate dynamic call graphs. Chapter 4 describes the overall design of SimSight and discusses implementation details of our SimSight prototype for x86/Linux environments. Chapter 5 reports the results of our experiments to evaluate the runtime overhead of SimSight using 12 benchmark programs from SPEC CPU2006 suite and our analysis of these results. Chapter 6 illustrates how SimSight can be used to detect software errors and increase program understanding. Chapter 7 highlights related research efforts that utilize virtual platforms to gather runtime information. We conclude the thesis and discuss future work in Chapter 8.

Chapter 2

Background

As mentioned in Chapter 1, the implementation of our proposed framework is based on Simics. In order to reconstruct the dynamic call graphs, we first need to capture the relevant procedure calls during simulation. We then analyze the captured information to reconstruct dynamic call graphs. In this chapter, we introduce relevant background information to facilitate understanding of the design and implementation of SimSight for x86/Linux executables. For readers who are familiar with these concepts, this chapter can be omitted.

2.1 Instructions Supporting Procedure Calls

Instruction set architectures provide different mechanisms to make procedure calls. As an example, MIPS architecture provides `JAL/JR` for procedure calls. On the other hand, x86 architecture provides `CALL/RET`. When a `CALL` instruction is executed, a new stack frame is created and the control flow is then transferred to the callee [5]. Figure 2.1 shows a common prologue sequence on x86. Assume that a *caller* is calling *func*.

```
call addr_of_func
push ebp        // save the old frame pointer
mov  ebp, esp   // get the new frame pointer
sub  esp, lsize // reserve place for locals in func
...
mov  esp, ebp   // free space for locals
pop  ebp        // restore the old frame pointer
ret             // return from the func
```

Figure 2.1: A Procedure Call Prologue

We capture every instance of `CALL` and `RET` instructions and stack information as a way to identify when a procedure call is made and when a procedure returns. More information about this process is provided in Chapter 4.

2.2 The Simics `Linux-Process-Tracker` and `Trace` Modules

Simics provides several loadable modules to support profiling and debugging. As examples, *g-cache* and *trace* are two popular modules used for profiling and tracing memory accesses. The source code of these modules is also available for customization.

A rich set of APIs is also provided to allow users to extend existed modules or design new modules. Users can specify the objects, interfaces, and events in a module using Python, C, C++, or the provided *Device Modeling Language.* Simics Command-Line Interface (CLI) also supports scripting (in Python or Simics scripting language) to facilitate automation. As part of our work, we utilize two existing Simics' modules: `linux-process-tracker (tracker)` and Simics `trace` module.

2.2.1 The Tracker Module

In multiprogramming operating systems such as Linux, we use the Simics' tracker, which is a kernel module that can track events occurring in a specific process running on a simulated Linux system. Currently, the tracker module can only work with Linux up to version 2.6.15 for PowerPC, UltraSPARC and x86 architectures. However, since the source code of the tracker is available, developers can create new trackers for different architecture/OS platforms.

2.2.2 The Simics Tracing Module

This module can be used to observe instruction fetches, data accesses, control register accesses, I/O accesses, and exceptions during the simulation. For our work, we designed a new module (*SimTracer*) to trace instructions of x86/Linux executables. More details about the design of SimTracer is presented in Chapter 4. Figure 2.2 shows how Simics tracker and trace module work during the simulation.

According to Figure 2.2, when the Simics tracker detects a specified executable name or Process ID, the tracing process is initiated. The output can be redirect to `stdout` or write to file. Note that we can trace multiple processes running in one processor or multiple processors.

2.3 Executable and Linkable Format

To construct the dynamic call graph of a binary program, SimSight needs to extract some information from the binary. Currently, *Portable Executable/Common Object File Format* (PE/COFF) [6] and *Executable and Linkable Format file format* (ELF) [7, 8] are two commonly used file formats for executables, object code, Dlls/shared libraries.

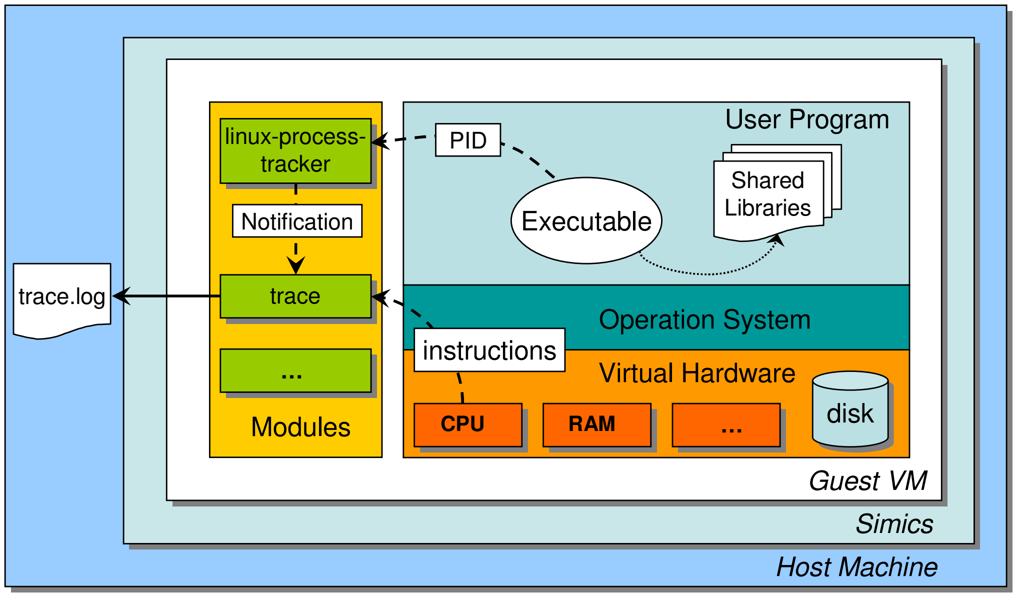


Figure 2.2: Tracker and tracer in Simics

PE/COFF is used in Windows operating system and ELF is used in Unix and Unix-like systems. There are three types of ELF objects: relocatable files, executable files, and shared objects. All the three objects share a similar file format, which include ELF Header, Program Header Table and Section Header Table. The ELF Header holds the information how to access to the Program Header Table and the Section Header Table. The Program Header Table contains the information that is used to create a process image. The Section Header Table describes the file's sections, which contain the instruction section, symbol table, and relocation information. We can get the section entry and section name from the Section Header Table.

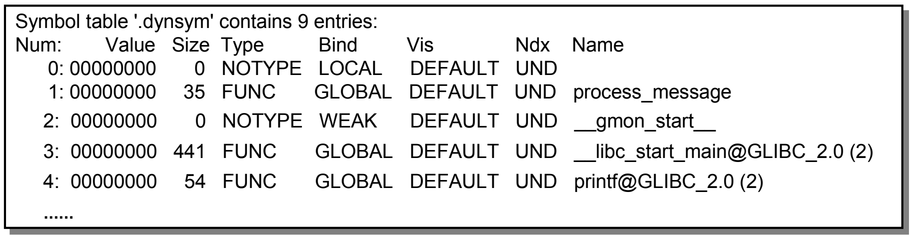
2.3.1 Static Linking and Dynamic Linking

In static linking, the static linker copies all library routines used in a program into an executable image during compilation. On the other hand, dynamic linking only places the name of shared libraries in an executable image. The *dynamic loader* then resolves these references at runtime [9, 10]. Shared objects in Linux are an example of dynamic linking. By using dynamic linking, executable files are smaller. However, the process to resolve references to dynamically linked functions is more complex than that of statically linked functions. More discussion is provided in Section 2.3.3.

2.3.2 The Symbol Tables

Dynamically linked executables and shared object usually have two symbol tables: `.SYMTAB` and `.DYNSYM`, which are used by the dynamic linker to locate or relocate the program's symbolic definitions and references. The `.SYMTAB` holds the debug information such as function names, function addresses, and binding types (`LOCAL/GLOBAL`) for both `LOCAL` and `GLOBAL` symbols. It is possible to remove `.SYMTAB` from an executable. In this scenario, it is not possible to extract the statically linked information.

The `.DYNSYM` section contains dynamic linking information that is used by the dynamic linker. Figure 2.3 is an example of the `.DYNSYM` section. The content is generated by GNU binary utility *readelf* [11]. In Figure 2.3, the "`UND`" means this symbol is not defined in this ELF object. It contains the information for the dynamic linker to relocate this symbol at runtime. As an example, function *process_message* has its function's address (symbol value) as `ZERO`; its symbol type as `FUNC` and its binding type as `GLOBAL`. In this case, this symbol must be resolved or relocated at runtime by the dynamic linker.



```
Symbol table '.dynsym' contains 9 entries:
Num:     Value  Size Type     Bind     Vis       Ndx  Name
   0: 00000000     0 NOTYPE   LOCAL    DEFAULT   UND
   1: 00000000    35 FUNC     GLOBAL   DEFAULT   UND  process_message
   2: 00000000     0 NOTYPE   WEAK     DEFAULT   UND  __gmon_start__
   3: 00000000   441 FUNC     GLOBAL   DEFAULT   UND  __libc_start_main@GLIBC_2.0 (2)
   4: 00000000    54 FUNC     GLOBAL   DEFAULT   UND  printf@GLIBC_2.0 (2)
   ......
```

Figure 2.3: An example of `.DYNSYM` section (executable object)

2.3.3 Relocation

Relocation is a process to resolve the addresses of dynamic linked functions, which are compiled as *position-independent code* (PIC) [12]. The mechanism is both platform- and OS-specific. In this thesis, we only discuss the relocation of Linux ELF on x86 architecture.

The ELF file format organizes program instructions, data, and auxiliary information into sections. There are two notable sections, `.PLT` and `.GOT` that are used in the relocation process. The `.PLT` section resides in `.TEXT` segment, and the `.GOT` section resides in the `.DATA` segment, allowing the dynamic linker to write to it during the relocation process.

Relocation is a two-step process involving both the static and dynamic linkers. During compilation, the static linker creates an entry for each dynamically linked function in the `.PLT` table of an executable. The destination address of each call instruction is represented as the index of the callee function in `.PLT`. At runtime, the dynamic linker substitutes these indexes with the actual addresses pointing to the `.PLT` entries. Thus, when a dynamically linked function is called, the processor jumps unconditionally to the entry, which performs either of the two tasks: jumping to the resolved address stored in the corresponding entry in `.GOT` or bootstrapping the relocation process if the address has not been resolved. For each of the unresolved function, the relocation routines in the dynamic linker are executed to resolve the address and patch the value into the `.GOT` entry. Such relocation is performed during the initial call to a dynamically linked function. Afterward, any call to this function can be done directly.

Chapter 3

Existing Approaches to Construct Dynamic Call Graphs

There are several existing approaches that can generate dynamic call graphs. Some of these employ source code instrumentation, while others work on executables. In this chapter, we focus specifically on approaches that (i) work on executables; (ii) are capable of tracking functions in dynamically linked libraries; and (iii) must be publicly available.

3.1 OS Integrated Tools

In this approach, instrumentation frameworks are built into operating system (OS) kernels. *DTrace*, an advanced dynamic tracing framework designed to improve the observability of software systems [13], is an example of this approach. Both Solaris and Mac OS have incorporated DTrace as a core component of their development and administration tools. DTrace enables users to observe the system by exporting various runtime *probes*, implemented and managed by *providers*. The *fbt* (*Function Boundary

Tracing) and *pid providers* support function tracing in the kernel and user-space. These *providers* allow tracing of any function entries and exits by attaching a trap immediately before each call instruction. DTrace is notified when this trap hits and automatically executes the user-defined *actions*. Because DTrace can instrument programs with low overhead, it is suitable for production environments.

Although such approaches are powerful and high-performance, they are tightly integrated with kernels and therefore, can only work in the kernels that support such features. Consequently, such tools do not work in a large class of embedded devices because they rarely use operating systems with such support.

3.2 OS Interface Tools

In this approach, tools are built to exploit OS and runtime interfaces to capture dynamic call graphs. As an example, *ltrace* or library trace is a debugging utility in Linux [14] that works with *fork* and *clone* system calls to perform function tracing. Currently, ltrace only intercepts the first function call to dynamically linked libraries. It traces neither the function calls between shared libraries nor statically linked function calls in programs. Moreover, ltrace only works in Linux.

To address some of these limitations, *latrace* extends ltrace to support tracing of dynamic function calls between shared libraries at runtime [15]. It is implemented on top of `LD_AUDIT`, which is the GNU dynamic linker audit feature. However, no dynamic library call can be traced if one of the shared libraries does not include a relocation *Procedure Linkage Table* (`.REL.PLT`) in the ELF binary. Both ltrace and latrace can operate with low overhead.

3.3 Dynamic Binary Instrumentation

Binary instrumentation can generate dynamic call graphs by inserting code snippets at the beginning of functions. However, in doing so, additional code is generated, which can result in some differences in runtime states when compared to native code with no instrumentation. As an example, *Pin* is an open-source binary instrumentation framework that has been widely used in debugging, profiling, and evaluating performance [16]. Pin provides several APIs so that developers can customize their own *Pintools* to perform tasks such as counting executed instructions and collecting function call information [16]. Currently, Pin can instrument Linux, Mac OS X, and Windows executables for several architectures. Recent work by Hazelwood and Klauser [17] shows that the overhead of Pin ranges from 1.5 to 8 times slower than native execution. Currently, Pin can support basic hardware devices, but it provides no functionality for developers to model their own devices. As such, its use in HW/SW co-design is still limited.

Another example is *Valgrind*, an instrumentation framework that can be used to build dynamic analysis tools. It currently works in Linux and Mac OS X. One of the tools in Valgrind that can be used to generate dynamic call graphs is *Callgrind*. It is an extension of *Cachegrind*, a cache profiler. Callgrind augments Cachegrind with call graph information so that it can generate call graphs for both statically and dynamically linked libraries [18]. The overhead of Callgrind ranges from 20 to 100 times slower than native execution.

3.4 Full System Simulators

Unlike typical instruction set simulators, which do not simulate I/O components, full-system simulators can be modeled to simulate complete computer systems with I/O components, bus interconnects, processors, and memory subsystems. Therefore, they provide virtual platforms that can run complex software systems (e.g., applications and OS kernels) without any modifications.

We conduct preliminary investigation to evaluate the suitability of two full system simulators, *QEMU* and Simics, as part of this work [19, 20]. In terms of performance QEMU is faster than Simics [21]. It also has *Trace Generation*, which is a component that works in conjunction with *DineroIV*, a memory reference tracing simulator, to generate execution traces and perform analysis [22]. However, QEMU lacks the capability to allow developers to model a full range of hardware devices. As such, QEMU is not as widely used in commercial HW/SW codesign projects as Simics [21].

Simics, on the other hand, provides infrastructure for developers to model and use hardware devices in their simulations. The modeling process is fast so engineers can have a new virtual platform up and running several months before the completion of the hardware prototype. As a commercial product, it also supports many advanced features and interfaces that developers can use to create their own instrumentation and dynamic analysis tools. However, there are currently no tools in the Simics toolkit that can generate dynamic call graphs. Based on our experience working with Simics, its execution overhead can range from 3 times (for processor intensive applications) to 30 times (for I/O intensive applications) slower than native execution.

3.5 Discussion

We believe that full-system simulators provide an attractive platform to carry out our work for three reasons.

3.5.1 Non-intrusive Instrumentation of executables

Instrumentation occurs at binary-level and without disturbing execution or affecting the virtualized state of a system. Therefore, it can simulate and profile systems accurately in the presence of instrumentation. Furthermore, these simulators can collect the exact profile data instead of relying on sampling or probability. Thus, the profiled information is more complete. For the problem we try to address, this is an important consideration.

3.5.2 Support more types of executables

Full-system simulators support executables with or without operating systems. This is different than other approaches, which are operating system dependent (e.g., Pin can only work on Linux or Mac OS X binaries). Therefore, our approach can work in diverse applications and systems ranging from executables running in stand-alone embedded devices with no operating systems to executables running in large computing clusters.

3.5.3 Popularity

HW/SW co-design is a widely adopted method to create computer systems. As such, full-system simulators especially Simics already play a prominent role in the development process. Therefore, developers who already use Simics for co-design can easily and effortlessly integrate the proposed extension as part of their testing and debugging toolkits.

In the next chapter, we discuss our implementation of SimSight, a dynamic call graph generator for Simics. We choose Simics over QEMU mainly due to its widespread adoption to support HW/SW co-design.

Chapter 4

Introducing *SimSight*

Because *Simics* can simulate different processor architectures and system configurations, the actual implementation of *SimSight* is system dependent. However, the overall approach is generic, i.e., there are essential components and steps required to implement a version of SimSight. We describe them in this chapter. We then discuss an actual implementation to support generating dynamic call graphs for x86/Linux executables.

We choose x86/Linux due to four major reasons. First, the execution model is more complex. Therefore, our implementation of SimSight must support many non-trivial features such as focusing on particular processes and supporting both statically and dynamically linked modules. Second, existing tools such as *latrace* can serve as an oracle to validate our dynamic call graph construction algorithm. Third, there are a large number of standardized benchmarks that can run on this platform. Fourth, our implementation to deal with the complexity of this platform would yield higher runtime overhead, which is likely to represent the worst-case overhead for our system.

We implemented SimSight on the Simics 4.0.40 simulator with x86-440bx-4.0.4 as the virtualized target. The host machine is equipped with a 2.66GHz Intel Core 2 Duo

CPU and 4GB of main memory. It runs Mandriva Linux with 2.6.27.24 multiprocessor kernel. The virtualized or guest machine is configured to run the *tango* image provided by Simics, which is based on Fedora Core 5 Linux with 2.6.15 kernel. The guest system is configured to be a single-core 2.2GHz with 256MB of memory. In typical Linux distributions, GNU dynamic linker (*ld*) and ELF are the default runtime linker and binary file format respectively. We also set up another system on a real machine to have a similar software configuration and environment to match the virtualized platform.

SimSight generates call graphs in three steps: (1) initial parsing of executables, which is accomplished by *SimAnalyzer*; (2) tracing of function call related instructions, which is accomplished by *SimTracer*; and (3) parsing the trace information to generate dynamic call graphs, which is accomplished by *SimAnalyzer* and *SimParser*. Next, we describe the main functionality and implementation of each of these three components.

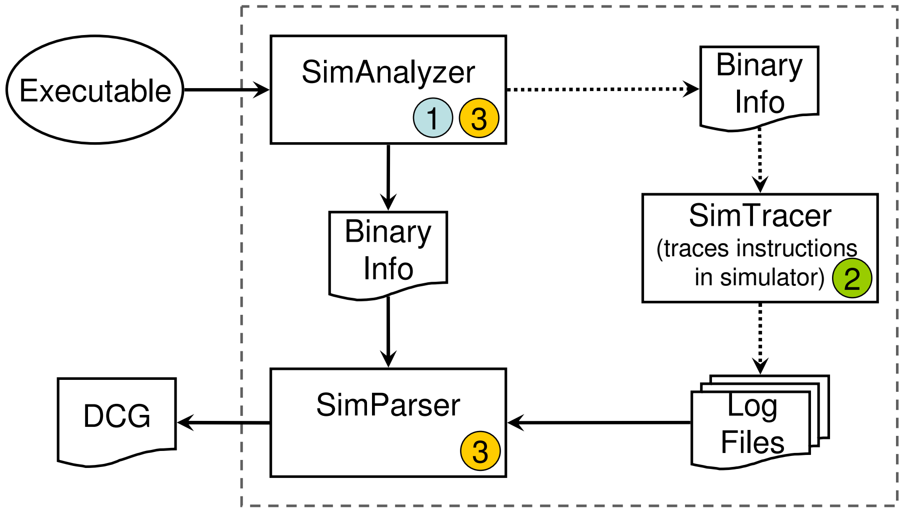


Figure 4.1: Basic workflow in SimSight. Note that circles with numbers indicate which component is used in a particular step in the workflow.

4.1 SimAnalyzer

SimSight is a symbolic function-tracing framework that operates on instruction traces generated by Simics virtual platform. Therefore, mapping of low-level information to high-level information is necessary. In most binary executable formats, storage and function information is provided to assist the dynamic linker and sometimes debugger. For example, both PE/COFF and ELF define a symbol table section embedded in the binary. The proposed SimAnalyzer is used to extract such pertinent information to perform this mapping.

Furthermore, dynamic linking systems such as that used in Linux also define relocation table, which contains information used to resolve addresses of dynamically linked functions. Our SimAnalyzer extracts the symbol names and addresses for statically and dynamically linked functions by parsing the symbol tables and relocation tables. This extracted information is stored for future use by SimTracer to filter out unwanted events and focus on wanted events.

In addition, SimAnalyzer also extracts information from dynamically linked libraries referenced by the main module. This information is used by SimParser to match late binding addresses recorded by our tracer to the actual symbolic function names as part of the call-graph construction. Thus, SimAnalyzer is first used to extract binary information for SimTracer (Step 1) in Figure 4.1. In addition, SimAnalyzer is also used during the parsing phase of our framework (shown as Step 3 in Figure 4.1).

4.1.1 Supporting x86/Linux Executables

Our analyzer extracts the address of each function from x86/Linux executables. For statically linked functions, the address can be easily found in the symbol table of that binary. Note that it is possible for the symbol table to be stripped from a Linux executable. This is not the case in Solaris, which requires this information to support tracing frameworks such as DTrace. Fortunately, in all executables that we evaluate, we find that symbol tables are still included. This is probably because part of the DTrace framework has been ported to Linux.

For dynamically linked functions in shared libraries, the address calculation is more complicated. In Linux, shared libraries are compiled as PIC, which can be mapped to any memory location prior to execution. These shared libraries are loaded as programs are being launched but the dynamically linked functions are resolved on demand by looking up in the *Procedure Linkage Table* (`PLT`) and patching the *Global Offset Table* (`GOT`). The algorithms to calculate function addresses are summarized below and illustrated in Figure 4.2 and Figure 4.3.

1. Statically Linked
    a) Main: `st_value`
    b) Library: `st_value + map_base`
2. Dynamically Linked
    a) Main: `plt_base + (si+1) * 16`
    b) Library: `plt_base + (si+1) * 16 + map_base`

For functions that are statically linked to the main module (1a), `st_value` is the value field of a function's entry in the symbol table (`.SYMTAB`). For statically linked functions to a library (1b), `map_base` is the memory address where the library is loaded. Figure 4.2 shows how the address resolved for statically linked functions.

For dynamically linked functions called from main or other libraries (2a) and (2b), `plt_base` is the start address of the `.PLT`; and `si` is the index of the function in

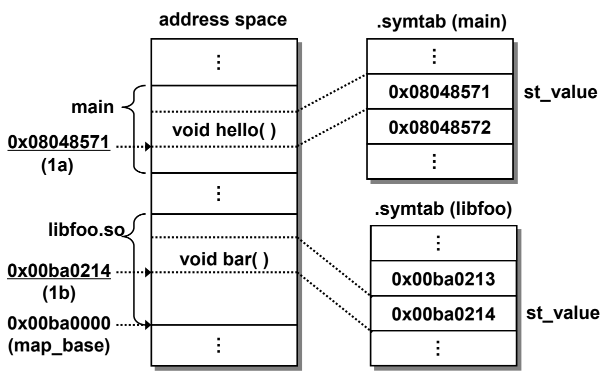


Figure 4.2: An illustration of calculating addresses of statically linked functions

the dynamic symbol table (`.DYNSYM`). The multiplier `16` is the size of each entry in the `.PLT` section. Figure 4.3 shows how the address resolved for dynamically linked functions.

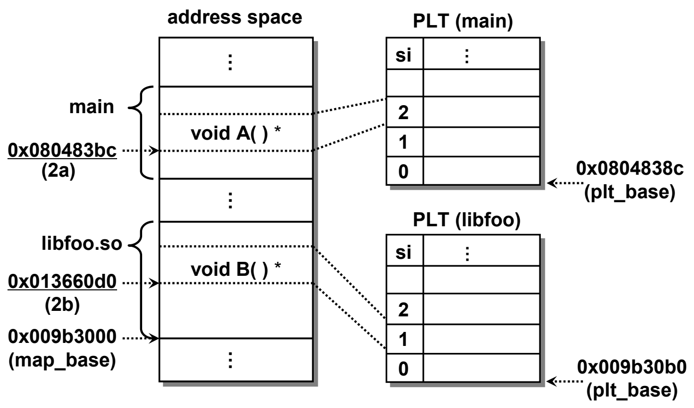


Figure 4.3: An illustration of calculating addresses of dynamically linked functions

The calculated addresses are then stored in a hash table, which is used by SimTracer and SimParser. In addition, the analysis tool also creates a data structure to store additional information such as symbolic name and resident library. Algorithm 1 describes the algorithm to implement SimAnalyzer for x86/Linux executables. Note that we construct the hash table `H` by extracting information based on the existence of symbol tables.

Algorithm 1 SimAnalyzer Algorithm

```
Require: program binary: bin
Ensure: hash table: H
  parse (.SYMTAB, .PLT, .REL.PTL, .DYNSYM)
  if .SYMTAB exists then
    Add(H, static/dynamic)
  else if .DYNSYM exists then
    Recalculate for dynamic (.PLT, .REL.PLT)
    Add(H, dynamic)
  else
    No ELF information
  end if
  return H
```

4.2 SimTracer

Most modern architectures have specialized instructions to support procedure calls. For instance, x86 and SPARC support procedure calls by the `CALL` instruction. Similarly, MIPS architecture offers the `JAL` (jump-and-link) instruction for making procedure calls. This, together with the advance in simulation technology, provides a golden opportunity for tracking any procedure call by targeting this type of instructions issued by virtual processors. Our proposed SimTracer is designed based on this simple idea.

In sophisticated virtual platforms such as Simics, dynamic tracing of memory accesses and instructions are already supported. Thus, our SimTracer is built on top of this feature to selectively monitor instructions related to procedure calls. However, filtering is also needed since some traced instructions do not always indicate function calls. As an example, in our x86/Linux implementation, instruction `PUSH` is only traced when it is used for address resolution of dynamically linked functions. Thus, we need to exclude the occurrences of `PUSH` for other purposes. Therefore, SimTracer evaluates information generated by SimAnalyzer to identify these extraneous instructions.

A major limitation of such trace-based approach is that the overhead is proportional to the amount of trace being generated. As will be shown later on, this overhead can sometimes be several orders of magnitude in very large programs. Furthermore, porting SimTracer to a new system (e.g., MicroC-OSII running on ARM9) will require thorough knowledge of function calling conventions of the architecture and possibly the OS.

4.2.1 Supporting x86/Linux Executables

SimTracer extends the functionality of the *trace* module provided by Simics. Its main responsibility is to capture and dissemble x86 instructions related to function calls. It also extracts other important runtime information from registers such as stack pointers, frame pointers, and privilege levels. The information for each application is continuously recorded into a trace log file, which is a compressed binary file. Figure 4.4 gives the structure of the trace log file.

`type`	`ebp`	`esp`	`eip`	`address`	`privilege`

Figure 4.4: Trace Log Format

In x86 architecture, stack is commonly used for parameter passing and the registers are used for returned values. This is due to the limited number of general-purpose registers. Typically, a procedure call is made through the `CALL` instruction. The stack frame is maintained using two special registers: `EBP` and `ESP`, which store the current *base frame pointer* and *stack pointer*.

According to the x86 calling convention, when a procedure call is made, the caller's frame pointer is pushed onto the stack and a new stack frame is created for the callee. Therefore, the current `EBP` will be the value of old `ESP` decremented by 4 bytes (in a 32-bit system) or 8 bytes (in a 64-bit system), and the current stack pointer points to the top of the stack. Note that in gcc x86 calling conventions, the stack grows downwards. When a procedure returns, the callee's stack frame is unwound and the saved caller's `EBP` is popped off the stack, which indicates the end of a procedure. Thus, we can detect procedure-call events by monitoring these two registers. Figure 4.5 shows the process of a procedure call represented in the stack frame of IA32.

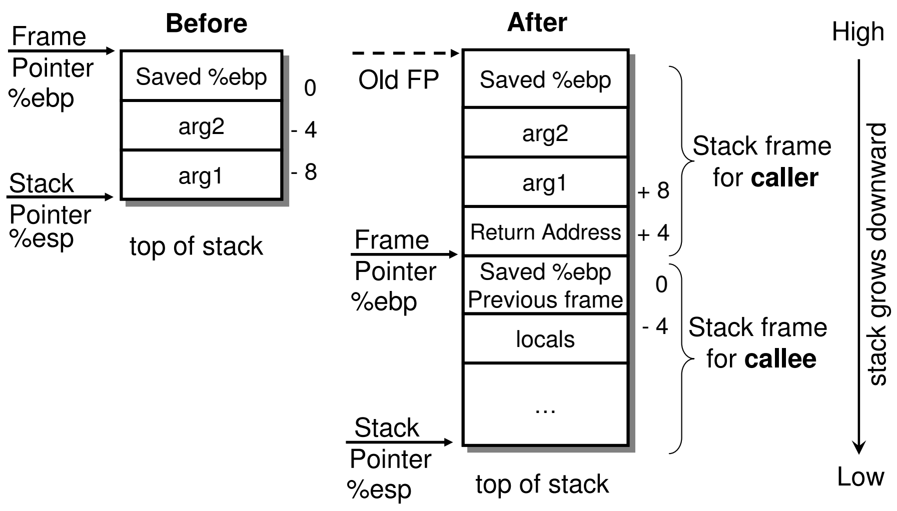


Figure 4.5: Stack Frame for Call Procedure

For dynamically linked functions in shared libraries, tracking invocation information is more difficult. This is because the target address of a procedure call instruction does not directly point to the real code of the invoked function. Instead, it initially points to "stub" code, which determines if the procedure has already been resolved. If it has not, the stub code invokes a runtime function to search for the shared library, patches the `GOT` table with the actual address of the dynamic function, and then performs an unconditional jump to that address. By monitoring registers, target addresses of function calls, and actual addresses of dynamic functions, our SimTracer can generate information that can be used to construct call graphs with complete calling contexts. The algorithm to implement SimTracer for x86/Linux executables shows in Algorithm

2.

Algorithm 2 SimTracer Algorithm

```
Require: program binary: bin
Require: prefix-set: CALL, PUSH EBP, RET
Ensure: trace log file: T.log
    symtab ← SimAnalyzer.read(bin)
    H ← SimAnalyzer.process(bin, symtab)
    if INST is in prefix-set then
        T.log ← Process Function Addresses(H, address)
    end if
```

Note that `Process Function Addresses(H, address)` is the aggregated information written to the trace log file. The `INST` is the instructions from the specified process running in the simulated system. The key consideration to generate this information is to monitor the contents of `EBP` and `ESP` registers that indicate occurrences of function calls as shown in Figure 4.5.

To automate the process, we developed a python script (*tracing.py*) that uses Simics APIs to issue commands through the Simics CLI. A handler is set up to register a *callback* for Simics tracker when a specified process is running. The callback then triggers SimTracer to trace related instructions defined by *prefix-set*. The tracer is activated until the process swapped out. The tracer is disabled when a different process is scheduled.

4.3 SimParser

Once the runtime traces have been generated, our proposed SimParser analyzes the trace files generated by the first two steps. The parsing process can be done either in a virtualized system or a real system. The latter is only applicable in the scenario that the real system has the same execution environment (exactly the same libraries, OS, etc.) as the virtualized system. This is to ensure library consistency of dynamically linked functions.

In our implementation, we choose to parse separately to reduce the parsing overhead. In this scenario, parsing can take very little time. On the other hand, if a system with the same runtime environment as the virtualized platform is not available, parsing has to be done in the virtualized platform. In this scenario, parsing can take much longer due to inefficient I/O and slower processor speed in simulators. Chapter 5 reports the parsing time for both scenarios.

4.3.1 Supporting x86/Linux Executables

The implementation of SimParser has two parts: extracting memory mapping information and dynamic call graph reconstruction. Next, we detail our implementations of these two functions.

Extracting memory mapping. In most UNIX-like operating systems, a `proc` pseudo file system exposes runtime information of each process. In addition, the file `/proc/<pid>/maps` reports the detailed memory mappings of the process identified by `<pid>` when queried. It is worth noting that there are alternative mechanisms such as LD_AUDIT [23], Library Interposer [24] for Linux and GNU runtime linker that can be used to obtain the same information. However, we did not use these approaches because they require calling additional methods that are not part of the traced programs, which can pollute the traced instructions. Instead, we developed a small program called *Spawn* to capture the `/proc/<pid>/maps` information when the binary is running in Simics. The information is saved to a file (map file) for later use by SimParser. Below is a snippet of the memory mappings of an example process.

```
address           perms offset  dev   inode  pathname
......
```

```
04442000-0463c000 r-xp 00000000 08:07 506739 /usr/lib/libpython2.6.so.1.0
0463c000-0463d000 r--p 001f9000 08:07 506739 /usr/lib/libpython2.6.so.1.0
0463d000-0468c000 rw-p 001fa000 08:07 506739 /usr/lib/libpython2.6.so.1.0
......
```

To trace a program, Spawn first launches the program as its child process by calling *posix_spawn* [25]. After getting the `pid`, Spawn reads the memory mapping of the target process from `/proc/<pid>/maps` and saves it to the user-specified log file.

Dynamic call graph reconstruction. As stated earlier, we leverage the relationship between base pointer or frame pointer (`EBP`) and stack pointer (`ESP`) to compute the level of each function in a calling hierarchy. Thus, SimParser maintains a *virtual stack* that is pushed and popped synchronously as the instruction traces are being processed. It follows the *x86/cdecl* calling convention[1]. Specifically, the parser memorizes the `EBP` and `ESP` of each call instruction on the virtual stack. As it parses to the next call, if both `EBP` and `ESP` are decremented (stack grows downward in x86), the parser concludes that the call is one level lower in the call graph and pushes its (`EBP`, `ESP`) onto the stack. If the `EBP` and `ESP` values of this function are greater than the top-of-stack value, it indicates multiple calls have returned. Thus, the parser also unwinds its virtual stack until it reaches the direct caller of the current callee. The caller function can be found by repeatedly comparing the current (`EBP`, `ESP`) with those on the virtual stack. Next, we present our algorithm used by SimParser to construct dynamic call graphs.

As shown in Algorithm 3, SimParser takes an executable as well as a trace log and library mapping, which were generated by SimTracer as inputs, returning the corresponding dynamic call graph `G`. SimAnalyzer is also used in this process to generate `symtab` and a function hash table (`H`) indexed by the runtime addresses of

[1]x86 calling conventions http://en.wikipedia.org/wiki/x86_calling_conventions

Algorithm 3 SimParser Algorithm

```
Require: trace log: log
Require: library mappings: maps
Require: program binary: bin
Ensure: dynamic call graph: G
    symtab ← SimAnalyzer.read(bin)
    H ← SimAnalyzer.process(bin, symtab)
    for lib in maps do
        symtab ← SimAnalyzer.read(lib)
        H ← SimAnalyzer.process(lib, symtab)
    end for
    for rec in log do
        level = convention(rec.ebp, rec.esp)
        G.add(H[rec.addr], level)
    end for
    return G
```

functions in both the main module and dynamic linked libraries. These data structures are used for mapping traced callee addresses to the actual function information.

Once the hash table is created, for each record in the trace log, we retrieve the base pointer `ebp` and stack pointer `esp` from each record. A platform-specific subroutine convention is invoked to compute the level of the procedure call with respect to the calling convention. Then the graph is repeatedly constructed by appending the function information `H[rec.addr]` with the computed level in the calling chain. In the end, the algorithm returns the dynamic call graph `G` to the user.

Without accounting for the time to retrieve symbol information from binaries, the *read* and *process* routines in SimAnalyzer have $O(n^2)$ time complexity, where n is the number of functions. Because on most platforms, the `convention` subroutine is *O(1)*; and in theory, hash table look up is also an *O(1)* operation, the total complexity of SimParser is bounded by $O(n^2)$.

Call graphs that are too complex may not provide the developers with sufficient insight to debug software problems. This is because complex call graphs are difficult to comprehend. Therefore, our implementation of SimParser for x86/Linux also includes a feature to allow developers to focus on particular modules and their interactions with other modules. In this mode, only function calls that originate from these modules are generated. We also provide commands so that developers can only focus on statically linked functions or dynamically linked functions. Our parser can also profile the number of invocations of each function in a program.

Chapter 5

Overhead of SimSight

In this chapter, we report results of our empirical evaluation to determine the overhead of SimSight. The evaluations were conducted on the system specified in Chapter 4. We used 12 benchmarks from SPEC CPU2006 Integer, a standardized CPU-intensive benchmark suite for evaluating performance of system's processor, memory subsystem and compiler. The first three columns in Table 5.1 describe each benchmark and its size, which ranges from 1.5K lines to over 250K lines.

There are three sources of overhead in our proposed framework. The first source is SimAnalyzer, which performs initial analysis of executable. The analysis is performed on a native machine so the amount of time needed to complete this process is negligible.

The second source is SimTracer, which is the most expensive component. According to Table 5.1, the overheads of SimTracer range from 3.5 to 28 times slower than execution times of Simics without it. The sizes of generated log files range from 120KB to about 1GB.

There are three major components in SimTracer. The first component is the tracing module, which is provided by Simics. By using this module, we find the execution to be 3 to 5 times slower than Simics without the tracing module. The

Benchmark	Description	LOC (K)	log size (MB)	W/O (seconds)	W (seconds)	Slowdown (times)
perlbench	Derived from Perl V5.8.7 interpreting various workload.	126	4.1	575.59	2280.14	3.96
bzip2	Julian Seward's bzip2 version 1.0.3 modified to do most work in memory.	5.73	8.3	347.65	1550.40	4.46
gcc	Based on gcc Version 3.2, generates code for Opteron.	236.27	61.8	274.81	2301.20	8.37
mcf	Uses a network simplex algorithm to schedule public transport.	1.57	7.2	181.36	2169.33	11.96
gobmk	Go plays the game of Go (AI).	157.65	1024	980.41	13799.93	14.07
hmmer	Protein sequence analysis using profile hidden Markov models.	20.66	13.6	387.04	2540.31	6.56
sjeng	A highly-ranked chess program that also plays several chess variants.	10.54	112.1	339.70	4358.4	12.83
libquantum	Simulates a quantum computer running Shor's factorization algorithm.	2.65	0.12	144.79	499.06	3.45
h264ref	An implementation of H.264/AVC encoding a videostream.	36	65.4	850.77	23894.40	28.09
omnetpp	Uses the OMNet++ simulator to model a large Ethernet network.	19.99	45	223.27	2086.44	9.34
astar	Pathfinding library for 2D maps.	4.28	60.5	296.88	8215.06	27.67
xalancbmk	Transforms XML documents to other document types.	267.32	23.8	172.16	743.58	4.32

Table 5.1: Describes the basic characteristic of each SPEC CPU2006 benchmark and reports the tracing overhead and size of each log file. Notes that W/O and W refers to SimTracer.

second component is the implementation of our algorithms to filter instructions and identify function addresses. The last component is to log the tracing information into a file. Figure 5.1 provides detailed distribution of the overhead based on these three runtime components.

As shown in Figure 5.1, in most applications, processing function addresses dominates the execution overhead. In each of the four benchmarks that has small slow-down (*libquantum*, *perlbench*, *bzip2*, and *xalancbmk*), the time spent by Simics' tracing module heavily contributes to the overhead. We also find that the logging component has very negligible effects on the overall overhead even when the size of the compressed log file is over 1GB. For the two applications with the greatest slow-down (*h264ref* and *astar*), the cost of processing the function addresses dominates the overhead.

The last source of overhead is SimParser. As stated earlier, SimParser can run on a native system if it is configured to be the same as the virtualized system. On

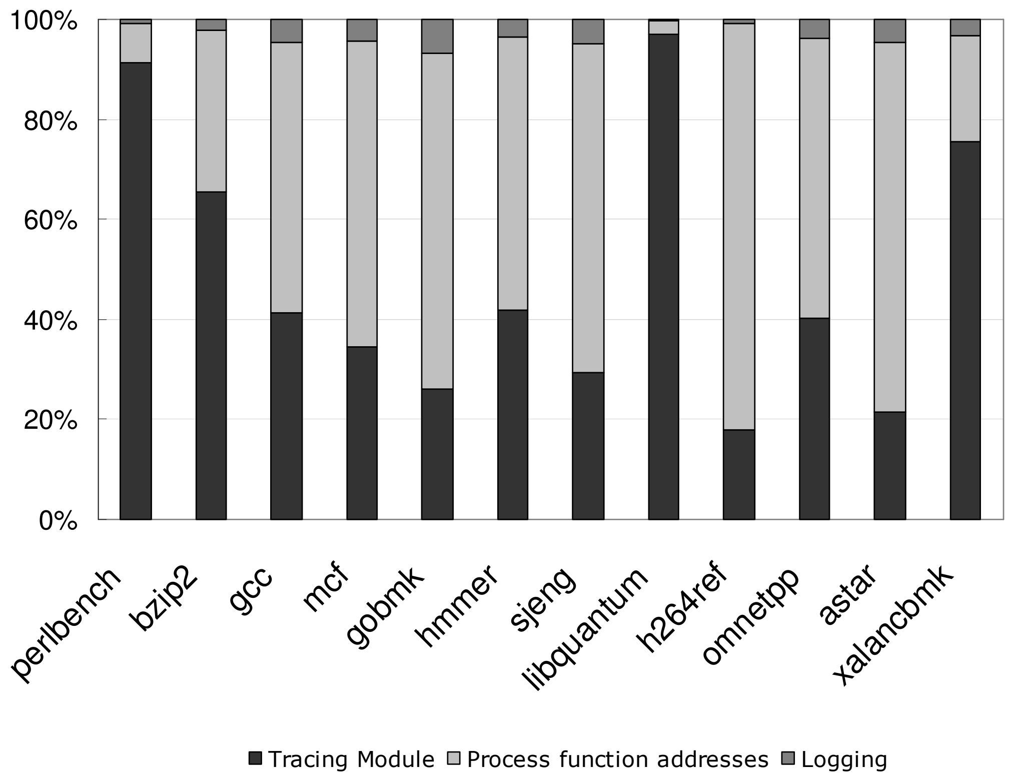


Figure 5.1: Overhead distribution of SimTracer for each benchmark.

the other hand, if such a native system is not available, SimParser must run in the virtualized system, which can result in much longer parsing time.

Running in a virtualized system. We first ran SimParser in the same virtualized system that we used to run SimTracer. It takes 61 seconds to parse the trace log of *libquantum* (less than 1 MB of compressed information), 16 minutes to parse the log of *perlbench* (4 MB), 61 minutes to parse the log of *h264ref* (65 MB), and nearly 3 hours to parse the log of *gobmk* (1 GB). These long parsing times are mainly due to inefficient file I/O in Simics.

Running in a real system. We performed the SimParser in a real system, which has a similar environment to that of the virtualized system. The performance of SimParser is 10 to 73 times faster than that of the virtualized execution. It only takes 1 second, 13 seconds, 6 minutes, and 17 minutes to parse the logs of *libquantum*, *perlbench*, *h264ref*, and *gobmk*, respectively. Table 5.2 listed the performances of the four benchmarks in these two systems.

Host/Guest	**libquantum**	**perlbench**	**h264ref**	**gobmk**
Trace log size	< 1 MB	4 MB	65 MB	1 GB
Virtualized System	61 seconds	16 minutes	61 minutes	3 hours
Real System	1 second	13 seconds	6 minutes	17 minutes

Table 5.2: SimParser Performance in virtualized / real system

In summary, parsing time is not a major performance factor if there is a system with a similar runtime environment to that of the virtualized system. On the other hand, if such a system is not available, the parsing time of a large log file can be many hours longer than that of the real machine.

Chapter 6

Usability Studies

In this chapter, we evaluate the usefulness of SimSight to aid developers to increase program understanding and isolate sources of programming errors.

6.1 Improved Program Understanding

Modern computer systems often employ advanced runtime systems to perform tasks that can make execution faster [26], overcome binary incompatibilities [27], ease the management of computing resources [28, 29], and increase programmer productivity [30]. For example, dynamic translator can be used to dynamically translate code written for one architecture to another architecture or dynamically optimize an executable that may have been compiled for a previous processor architecture [31, 27]. Runtime systems such as Hardware Abstraction Layers and garbage collectors are used to simplify the management of hardware components and memory, accordingly. Dynamic linkers and loaders are used to simplify the task of managing shared binary objects such as libraries that are commonly used by applications [30, 9, 10, 32].

On the other hand, these runtime systems can make understanding program execution more difficult. This is because their executions often interleave with the application execution. For example, reference counting, an automatic dynamic memory management technique used in Perl and Visual Basic, performs accounting tasks to track changes in the object reference graph and increments and decrements references while a program is running [29]. A dynamic linker and loader loads a dynamically linked object the first time it is accessed [32]. It also needs to update the reference so that subsequent accesses can be done directly. In these two examples, the time spent executing these runtime systems accounts for part of the overall execution time of an application. Furthermore, many of these runtime systems can change application behaviors (e.g., less efficient memory management techniques can suffer from out-of-memory errors or cause memory leaks). Therefore, understanding when and how often these runtime systems are invoked can be helpful to application developers who have to test and debug these systems.

In SimSight, any procedure call information related to an application is automatically captured. This information includes any calls made by the application and calls made by runtime systems to support the application. For example, our implementation can log any calls to dynamic linker and loader (calls to `ld.so`) and any calls made by `ld.so` to its helper functions, allowing developers to observe module and function-call dependencies. Figure 6.2 shows the dynamic call graph of a program. Note that functions *_dl_fixup*, *_dl_lookup_symbol_x*, and *do_lookup_x* are used to access dynamically linked functions.

6.2 Improved Debugging Capability

It is a common practice for software developers to rely on existing software components to perform some tasks for their applications. Unfortunately, these common components often have their own complex module dependencies that can result in hard-to-understand and possibly conflicting module dependencies. Thus, maintaining library compatibility in systems that use dynamic link libraries is a challenging problem [30].

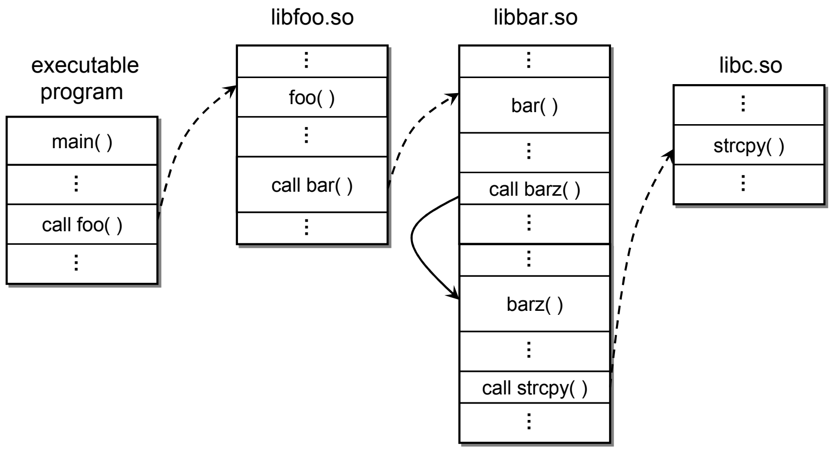


Figure 6.1: Dependency of our sample program

Figure 6.1 depicts a simple example that utilizes three shared libraries: `libfoo.so`, `libbar.so`, and `libc.so`. Note that a solid arrow represents a call to a function local to the libary. dotted arrow represents a call to a dynamically linked function. As shown in the figure, `main()` calls `foo()`, which is part of the shared library `libfoo.so`. Within `foo()`, there is a function call to `bar()`, which is part of the shared library `libbar.so`. In `bar()`, there is a static function call to `barz()`, which is also in `libbar.so`, and within `barz()`, there is a call to `strcpy()`, which is part of the shared library `libc.so`. In this dependency structure, it might be possible that a `libc.so` upgrade can cause this program to fail. In this example, when we try to execute this program, a “segmentation fault” occurs.

Typically, this type of the error message appears without providing the precise location that causes the error. One option is to use SimSight to generate a call graph that includes both statically and dynamically linked libraries. Figure 6.2 shows the result from SimSight. Note that it reports every function call made by the program.

```
__libc_start_main [/lib/libc-2.4.so] [D]
_dl_fixup [/lib/ld-2.4.so] [S]
  _dl_lookup_symbol_x [/lib/ld-2.4.so] [S]
    do_lookup_x [/lib/ld-2.4.so] [S]
      strcmp [/lib/ld-2.4.so] [S]
      ...
      foo [/usr/lib/libfoo.so.1.0] [D]
      _dl_fixup [/lib/ld-2.4.so] [S]
        _dl_lookup_symbol_x [/lib/ld-2.4.so] [S]
          do_lookup_x [/lib/ld-2.4.so] [S]
            strcmp [/lib/ld-2.4.so] [S]
            strcmp [/lib/ld-2.4.so] [S]
            strcmp [/lib/ld-2.4.so] [S]
        bar [/usr/lib/libbar.so.1.0] [D]
        _dl_fixup [/lib/ld-2.4.so] [S]
          _dl_lookup_symbol_x [/lib/ld-2.4.so] [S]
            do_lookup_x [/lib/ld-2.4.so] [S]
              strcmp [/lib/ld-2.4.so] [S]
              ...
              strcmp [/lib/ld-2.4.so] [S]
          0x0049e1dc [-] [-]
          _dl_fixup [/lib/ld-2.4.so] [S]
            _dl_lookup_symbol_x [/lib/ld-2.4.so] [S]
              do_lookup_x [/lib/ld-2.4.so] [S]
                strcmp [/lib/ld-2.4.so] [S]
                ...
          barz [/usr/lib/libbar.so.1.0] [S]
            strcpy [/lib/libc-2.4.so] [D]
            _dl_fixup [/lib/ld-2.4.so] [S]
              _dl_lookup_symbol_x [/lib/ld-2.4.so] [S]
                do_lookup_x [/lib/ld-2.4.so] [S]
                  strcmp [/lib/ld-2.4.so] [S]
                  ...
                  strcmp [/lib/ld-2.4.so] [S]
```

Figure 6.2: Dynamic Call Graph of our sample program

In summary, developers can use SimSight to assist with identifying difficult errors such as library incompatibility and achieve greater system observability.

Chapter 7

Related Work

In academic research, virtual platforms such as Simics are often used to simulate new research ideas in hardware. For example, numerous researchers have used virtual platforms to model new memory organization or cache optimization [33, 34, 35] then evaluate their effectiveness by running benchmark programs in the virtual platforms [36, 37].

In addition, researchers also use these virtual platforms to observe low-level runtime behaviors that can be difficult to obtain in real hardware. For example, Wright et al. [37] uses Simics to observe the behaviors of the HotSpot JVM. The goal is to be able to achieve non-disruptive inspection of the JVM states. As part of this work, they create a service that can relate low-level events back to JVM activities. For example, they create a service module that can map virtual addresses to symbolic names. Li et al. uses full system simulation to characterize the behaviors of SPECjvm98, a standardized Java benchmarks at that time [35]. In their work, they profile execution time of JIT compiler and interpreter, cache behavior, paging behavior, and instruction-level parallelism characteristic.

Albertson introduces *Holistic Debugging* as a method for observing execution of complex software systems [38]. Simics was used to non-intrusively gather low-level runtime information. The holistic debugger framework then maps this low level information to higher abstraction level observation tools such as debugger. A prototype was built on Simics to map low level storage information to application level data such as variables and types. This is accomplished by parsing the data structures of the operating system and virtual platform. One major difference between the standard debugger and holistic debugger is the use of Simics non-intrusive probing to support debugging. Our work shares a similar motivation with that of [38, 37]; that is, we want to take advantage of the non-intrusive probing and execution observability in virtual platforms to improve software quality. As such, it may be acceptable to suffer significant runtime overhead in favor of greater visibility and completeness.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

We have described SimSight, a framework for generating dynamic call graph based on virtualization. The motivation for introducing the proposed framework is to take advantage of the non-intrusive probing and execution observability in virtual platforms to improve software quality. This is done in spite of suffering significant runtime overhead. We then implement a prototype in Simics full system simulator to support x86/Linux executables and apply it on a case study. To evaluate the overhead to capture function call information, we use 12 programs from SPEC CPU2006 benchmark suite. The result indicates that the execution times of Simics with function tracing is 3.5 to 28 times slower than those of Simics with no tracing. The parsing time is also not significant if it can be accomplished in a real system.

8.2 Future Work

For future work, we plan to experiment with on-demand tracing to reduce overhead and increase applicability. For example, a scenario that can benefit from on-demand tracing mode is when a program crashes after a certain period of execution or when a certain human detectable event occurs. In this situation, the subset of the call graphs leading up to the failure may be more interesting. To support on-demand tracing, we exploit the snapshot feature of Simics. A snapshot is a set of files that contain enough information about the system and the processes running on the system to enable restart [39, 40]. In the on-demand tracing mode, SimSight begins tracing from the snapshot location. By starting close to an execution point of interest, we can eliminate the unnecessary tracing efforts while still generating useful module interaction information. We will also explore an event-based technique to initiate tracing.

Bibliography

[1] Yazarel, H., Kaga, T., Butts, K.: High-confidence powertrain control software development. In: SCDAC. (2006)

[2] Deviceguru.com: Over 4 Billion Embedded Devices Ship Annually. http://deviceguru.com/over-4-billion-embedded-devices-shipped-last-year/ (2008)

[3] Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hällberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35** (2002) 50–58

[4] Takalo, J., Kaariainen, J., Parviainen, P., Ihme, T.: Challenges of Software-Hardware Codesign. White Paper (2007) http://www.vtt.fi/inf/pdf/workingpapers/2008/W91.pdf.

[5] Bryant, R.E., O'Hallaron, D.R.: Computer Systems: A Programmer's Perspective. Prentice Hall, Upper Saddle River, NJ, USA (2002)

[6] Microsoft: (Microsoft portable executable and common object file format specification)

[7] Unix System Laboratories: (Executable and linkable format)

[8] Bahrami, A.: Elf symbol tables. (http://blogs.sun.com/ali/entry/inside_elf_symbol_tables)

[9] Levine, J.R.: Linkers and Loaders. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)

[10] Levine, J.R.: Linkers and loaders. (http://www.iecc.com/linker/)

[11] GNU: readelf. (http://www.gnu.org/software/binutils/)

[12] Gentoo Linux: Introduction to position independent code. (http://www.gentoo.org/proj/en/hardened/pic-guide.xml)

[13] Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2004) 2–2

[14] Haas, J.: What is ltrace. (http://linux.about.com/cs/linux101/g/ltrace.htm)

[15] freshmeat: latrace. (http://freshmeat.net/projects/latrace)

[16] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2005) 190–200

[17] Hazelwood, K., Klauser, A.: A dynamic binary instrumentation engine for the arm architecture. In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Seoul, Korea (2006) 261–270

[18] Weidendorfe, J.: Kcachegrind - call graph viewer. (http://kcachegrind.sourceforge.net/html/Home.html)

[19] Bellard, F.: QEMU, a fast and portable dynamic translator. In: ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2005) 41–41

[20] Virtutech: Wind river simics - embedded system simulation platform. (http://www.virtutech.com/)

[21] PTLsim: Background: Virtual machines and full system simulation. http://www.ptlsim.org/Documentation/html/node16.html (2010)

[22] Edler, J., Hill, M.: Dinero IV: Trace-driven uniprocessor cache simulator. (http://pages.cs.wisc.edu/ markhill/DineroIV)

[23] GNU: rtld-audit - auditing api for the dynamic linker. (http://www.kernel.org/doc/man-pages/online/pages/man7/rtld-audit.7.html)

[24] Nakhimovsky, G.: Debugging and performance tuning with library interposers. http://developers.sun.com/solaris/articles/lib_interposers.html (2001)

[25] The IEEE and The Open Group: (posix_spawn) http://www.opengroup.org/onlinepubs/009695399/functions/posix_spawn.html.

[26] Hazelwood, K., Smith, M.D.: Managing bounded code caches in dynamic binary optimization systems. ACM Trans. Archit. Code Optim. **3**(3) (2006) 263–294

[27] Apple Inc.: Apple Rosetta. http://www.apple.com/rosetta (2007)

[28] Altera Corp.: Overview of Hardware Abstraction Layer. On-line Documentation (Last retrieved: June 10) http://www.altera.com/literature/hb/nios2/n2sw_nii52003.pdf.

[29] Jones, R., Lins, R.: Garbage Collection: Algorithms for automatic Dynamic Memory Management. John Wiley and Sons (1998)

[30] Donald, J.: Improved portability of shared libraries. http://www.princeton.edu/ jdonald/research/shared_libraries/cs518_report.pdf (2003)

[31] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S.B., Yates, J.: Fx!32 - a profile-directed binary translator. IEEE Micro **18** (1998) 56–64

[32] : ld-linux(8) - linux man page. (http://linux.die.net/man/8/ld-linux)

[33] Yoon, D.H., Erez, M.: Memory mapped ecc: low-cost error protection for last level caches. In: ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture, Austin, TX, USA (2009) 116–127

[34] Magnusson, P., Werner, B.: Efficient memory simulation in simics. Simulation Symposium, Annual **0** (1995) 62

[35] Li, T., John, L.K., Narayanan, V., Sivasubramaniam, A., Sabarinathan, J., Murthy, A.: Using complete system simulation to characterize specjvm98 benchmarks. In: ICS '00: Proceedings of the 14th international conference on Supercomputing, Santa Fe, New Mexico, United States, ACM (2000) 22–33

[36] Schindewolf, M.: Analysis of Cache Misses Using Simics. PhD thesis, University of Karlsruhe, Karlsruhe, Germany (2007)

[37] Wright, G., McGachey, P., Gunadi, E., Wolczko, M.: Introspection of a Java Virtual Machine under simulation. Technical report, Mountain View, CA, USA (2007)

[38] Albertsson, L.: Holistic debugging – enabling instruction set simulation for software quality assurance. Modeling, Analysis, and Simulation of Computer Systems, International Symposium on **0** (2006) 96–103

[39] Tuthill, B., Johnson, K., Schultz, T.: Irix checkpoint and restart operation guide. On-Line Manual (2003) http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=bks&fname=/SGI_Admin/CPR_OG/front.html.

[40] Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent Checkpointing under Unix. Technical report, Knoxville, TN, USA (1994)