2010

SimSight: A Virtual Machine Based Dynamic Call-Graph Generator

Xueling Chen
University of Nebraska-Lincoln, xchen@cse.unl.edu

Peng Du
University of Nebraska-Lincoln, pdu@cse.unl.edu

Witawas Srisaan
University of Nebraska-Lincoln, witty@cse.unl.edu

Department of Computer Science & Engineering, University of Nebraska-Lincoln,
Technical Report, TR-UNL-CSE-2010-0010

SimSight: A Virtual Machine Based Dynamic Call-Graph Generator

Xueling Chen, Peng Du, and Witawas Srisa-an

Computer Science and Engineering
University of Nebraska-Lincoln
256 Avery Hall
Lincoln, NE 68588-0115
xchen@cse.unl.edu, pdu@cse.unl.edu, witty@cse.unl.edu

Abstract

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. As such, 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*, which is a widely adopted commercial full-system simulator. *Simics* is a software system that simulates complete computer systems. As such, 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. We then evaluate its performance using 12 integer programs from SPEC CPU2006 benchmark suite.

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 an 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 their expertise to build specific software components that are reusable.

On the other hand, such practice can also lead to some dependability issues especially in systems that reuse many software components or modules 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 [41]. 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 [41]. 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 drive-train 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 [41].

Obtaining a complete view of module interactions is challenging in this scenario 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' drive-train 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 systems utilizing unsupported or no 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 [9]. 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, such binary instrumentation tools only support a subset of processor architectures and operating systems.

Additionally, most of existing tools do not provide infrastructure for device modeling. As such, 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 [41]. 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 to enhance software dependability.

This work. We explore the use of a full-system simulator, *Simics*, as a way to provide module interactions in the form of dynamic call graphs (DCGs). 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) 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; (iii) due to 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 [36]; and (iv) licensing of Simics is free for academic institutions, making it a good platform for research.

In the first part of this work, we describe the implementation details of *SimSight*, our proposed framework to capture dynamic call graphs in Simics. This includes the implementations of both the tracing and parsing functions. In the second part, we evaluate the cost of generating dynamic call graphs using the proposed SimSight. We employ SimSight in a complex computing environment using an advanced operating system and 12 non-trivial benchmarks. We then compare the execution time of each benchmark on SimSight with the execution time on the actual system, and the execution time on Simics without SimSight. When compared to the unmodified Simics, SimSight takes 4 to 28 times longer to execute. As such, it can incur the overheads ranging from 27 to 475 times slower than executing the benchmark programs on the actual system.

The remainder of this paper is organized as follows. Section 2 summarizes existing tools that can generate dynamic call-graphs. Section 3 describes the overall design of SimSight and discusses implementation details of our SimSight prototype for x86/Linux environments. Section 4 reports the results of our experiments to evaluate the runtime overhead of *SimSight* using 12 benchmark programs from SPEC CPU2006 suite and our analyses of these results. Section 5 provides a usability study using two examples. Section 6 highlights related research efforts that utilize virtual platforms to gather runtime information. Section 7 discusses our plan to improve SimSight.

2. Approaches to Construct Dynamic Call Graphs

A dynamic call graph is a record of a program execution [34]. Dynamic call graphs have played an important role in helping software developers test and debug their programs[16, 33]. As an example, dynamic call graphs can be used to measure function coverage, identify unreachable functions [14], and determine code dependency [12]. Furthermore, they can help with program understanding [29].

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 section, 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. Next, we describe the capabilities of these approaches.

2.1 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 [7], is an example of this approach. Both Solaris and Mac OS have incorporated DTrace as a core component on their development and administration tools. DTrace enables users to observe system behaviors 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.

2.2 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 [17] 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 [13]. 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.

2.3 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 [25]. 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 [25]. Currently, Pin can instrument Linux, Mac OS X, and Windows executables for several architectures. Recent work by Hazelwood and Klauser [18] 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 codesign 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 [39]. The overhead of Callgrind ranges from 20 to 100 times slower than native execution.

2.4 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 [6, 38]. In terms of performance *QEMU* is faster than Simics [32]. It also has *Trace Generation*, which is a component that works in conjunction with *Dinero IV*, a memory reference tracing simulator, to generate execution traces and perform analysis [11]. 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 [32].

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 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.

2.5 Discussion

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

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. As such, the profiled information is more complete. For the problem we try to address, this is an important consideration.

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). As such, 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.

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. As such, developers who already use Simics for codesign can easily and effortlessly integrate the proposed extension as part of their testing and debugging toolkits.

In the next section, 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.

3. Introducing SimSight

The goal of SimSight is to provide greater observability at the cost of degraded performance. We believe that this trade-off is acceptable since we envision that SimSight will be used during the software development process and not in deployed environment. Furthermore, because Simics is already widely used to support software development during HW/SW codesign period, developers should be accustomed to the practice of achieving functional correctness and not absolute performance by using such a development tool.

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 section. We then discuss an actual implementation to support generating dynamic call graphs for x86/Linux executables. We choose x86/Linux due to the following four reasons.

1. The execution model of Linux is more complex. As such, our implementation of SimSight must support many non-trivial features such as focusing on particular processes and supporting both statically and dynamically linked modules.
2. Existing tools such as *latrace* can serve as an oracle to validate our dynamic call graph construction algorithm.
3. There are a large number of standardized benchmarks that can run on this platform.
4. Our solutions 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.

Note that we have also implemented a version of SimSight that works with MicroC/OS-II real-time operating system. In this implementation, MicroC/OS-II is running on MS-DOS, which acts as the resource and I/O manager for MicroC/OS-II. However, there are not many publicly available benchmarks that can be used for performance evaluation. As such, it is not evaluated in this paper.

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 call graphs, which is accomplished by SimAnalyzer and *SimParser*. Next, we describe the main functionality and implementation of each of these three components.

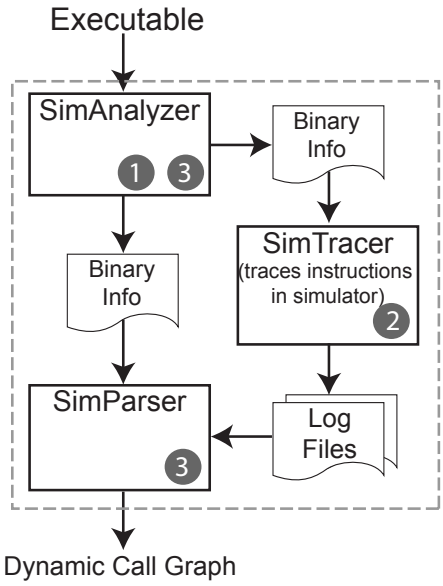

Figure 1: Basic workflow in *SimSight*. Note that gray circles indicate which component is used in a particular step in the workflow.

3.1 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 *Portable Executable/Common Object File Format* (PE/COFF) [27] and *Executable and Linking Format* (ELF) [21, 5] 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 call-graph construction. As such, SimAnalyzer is first used to generate binary information for SimTracer (Step 1) in Figure 1. In addition, SimAnalyzer is also used during the parsing phase of our framework (shown as Step 3 in Figure 1).

3.1.1 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.

For dynamically linked functions in shared libraries, the address calculation is more complicated. In Linux, shared libraries are compiled as *position independent code* (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 2.

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 (`.sym`). For statically linked functions to a library (1b), `map_base` is the memory address where the library is loaded. For dynamically linked functions called from main or other libraries (2a) and (2b), `plt_base` is the start address of the *procedure linkage table* (`plt`); and `si` is the index of the function in the dynamic symbol table (`.dynsym`). The multiplier `16` is the size of each entry in the `plt` section. The calculated addresses are then stored in a *Hash Table*, which will then be 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.

3.2 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. As such, 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. As such, 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.

3.2.1 Supporting x86/Linux Executables.

SimTracer extends the functionality of the *tracer* module provided by Simics. The module is responsible for intercepting and dissembling 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 compressed binary file.

In x86 architecture, stack is commonly used for parameter passing and the registers are used for returned values. This is due to limited number of general-purpose registers. In x86, a typical procedure call is made through the `call` and `ret` instructions. 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 (32-bit) or 8 (64-bit) bytes, and current stack pointer points to the top of the stack. 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.

For dynamically linked functions in shared libraries, tracking invocation information is more difficult. This is be-

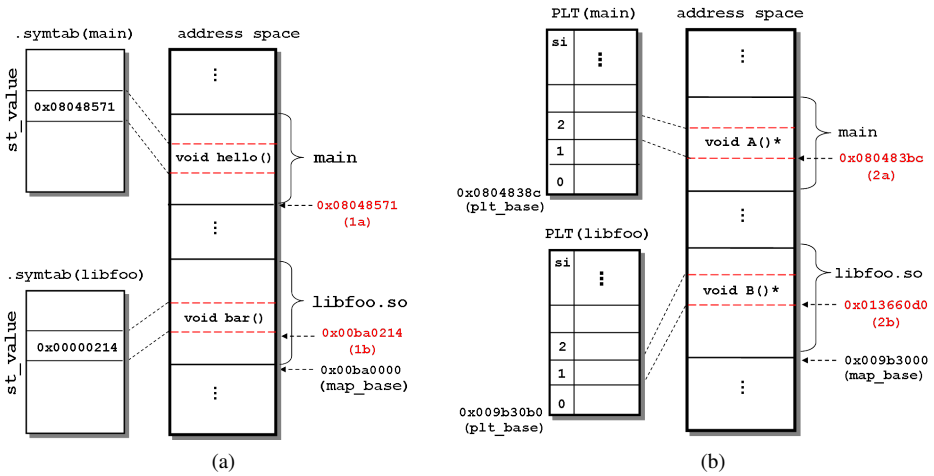

Figure 2: (a) illustrates address calculation for statically linked functions and (b) illustrates calculation for dynamically linked functions.

cause 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.

To automate the process, we create a *Python* script that uses Simics APIs to issue commands through the *Command-Line Interface* (CLI). The script sets up the Linux-process-tracker and SimTracer and registers *callbacks* for function-call related instructions. The Linux-process-tracker is a Simics-provided module that allows tracking user-specified processes by either *process id* (*pid*) or file name in Linux. If the running process is an instance of the target program, we enable the tracer until the next context switch. The tracer is disabled when a different process is scheduled. Linux-process-tracker is essential for SimTracer to select instruction traces from only the target process.

3.3 SimParser

Once the runtime traces have been generated, our proposed SimParser is used to process the information. By following the static and dynamic calling conventions of the simulated system (e.g., processor-specific calling conventions or OS-specific dynamic linking convention), SimParser constructs dynamic call graphs by analyzing the values of instruction pointers and stack frame updates. Note we can perform parsing during tracing or as a separate process.

In our implementation, we choose to parse separately to reduce the tracing overhead. This is because tracing must be done in a virtualized platform; however, parsing can be done on a real system that has been configured to have the same runtime environment as the virtualized platform. This similarity is required to ensure library consistency of dynamically linked functions. 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. Section 4 reports the parsing time for both scenarios.

3.3.1 Supporting x86/Linux Executables.

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` [15], *Library Interposer* [28] 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.

As stated earlier, we leverage the relationship between base pointer (`ebp`) and stack pointer (`esp`) to compute the level of each function in the 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

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

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` at this function are greater than the top-of-stack value, it indicates multiple calls have returned. Thus, the parser also unwinds its virtual stack by popping the 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.

Algorithm 1 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
```

As shown in Algorithm 1, 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 runtime addresses for 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 `bp` and stack pointer `sp` 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. Based on this algorithm, a snippet of call-graph generation is presented in Figure 3.

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

```
printf (0x80483a4) [/lib/libc-2.4.so]
_dl_fixup (0x9a350b8) [/lib/ld-2.4.so]
  __i686.get_pc_thunk.bx (0x9ab1db) [/lib/ld-2.4.so]
  _dl_lookup_symbol_x (0x99f44c) [/lib/ld-2.4.so]
    __i686.get_pc_thunk.bx (0x9ab1db) [/lib/ld-2.4.so]
    do_lookup_x (0x99f12b) [/lib/ld-2.4.so]
      __i686.get_pc_thunk.bx (0x9ab1db) [/lib/ld-2.4.so]
      strcmp (0x9aad00) [/lib/ld-2.4.so]
      strcmp (0x9aad00) [/lib/ld-2.4.so]
      _dl_name_match_p (0x9a4210) [/lib/ld-2.4.so]
        __i686.get_pc_thunk.bx (0x9ab1db) [/lib/ld-2.4.so]
        strcmp (0x9aad00) [/lib/ld-2.4.so]
        strcmp (0x9aad00) [/lib/ld-2.4.so]

... Omitted ...

  vfprintf (0x9ed0f9) [/lib/libc-2.4.so]
    __i686.get_pc_thunk.bx (0x9c8650) [/lib/libc-2.4.so]
    __find_specmb (0xa06564) [/lib/libc-2.4.so]
      __i686.get_pc_thunk.bx (0x9c8650) [/lib/libc-2.4.so]
      __i686.get_pc_thunk.bx (0x9c8650) [/lib/libc-2.4.so]
        __i686.get_pc_thunk.bx (0x9c8650) [/lib/libc-2.4.so]
        _IO_doallocbuf (0xa12bf2) [/lib/libc-2.4.so]
```

Figure 3: A snippet of call-chain information

3.4 Additional Features

As stated earlier, our host and guest systems are not exactly the same. By default Linux programs linked shared libraries dynamically. As such, our parser needs to run in the guest system to locate the correct libraries for analyzing. On the other hand, if a native system is set up to be identical in terms of kernel, libraries, and processor architecture to the guest system, it is possible to run SimParser natively, lowering the parsing time.

Besides constructing dynamic call graphs, our implementation of SimSight also supports invocation counting and privilege auditing as two extra features. We can accomplish the latter feature by extending SimTracer to record privilege level information. The information is then used to detect *privilege escalation* attacks by malicious software [31].

3.5 Limitations

Currently, our implementation only works with function call instructions (e.g., call, jal). It does not work with calls made by simple jump instructions. It also does not work with stripped binaries, in which symbol tables are not included.

4. Overhead of SimSight

In this section, we report results of our empirical evaluation to determine the overhead of SimSight. 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 an 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 512MB of main 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 native machine to have a similar software configuration and hardware environment to match the virtualized platform. This native machine is used to perform parsing. We then use 12 benchmarks from SPEC CPU2006 Integer, a standardized CPU-intensive benchmark suite for evaluating performance of system's processor, memory subsystem and compiler. Table 1 describes each benchmark and its size, which ranges from 1.5K lines to over 250K lines.

Benchmark	Description	LOC (K)
libquantum	Simulates a quantum computer running Shor's factorization algorithm.	2.65
perlbench	Derived from Perl V5.8.7 interpreting various workload.	126
h264ref	An implementation of H.264/AVC encoding a videostream.	36
gobmk	Go plays the game of Go (AI).	157.65
astar	Pathfinding library for 2D maps.	4.28
mcf	Uses a network simplex algorithm to schedule public transport.	1.57
hmmer	Protein sequence analysis using profile hidden Markov models.	20.66
gcc	Based on gcc Version 3.2, generates code for Opteron.	236.27
omnetpp	Uses the OMNet++ simulator to model a large Ethernet network.	19.99
bzip2	Julian Seward's bzip2 version 1.0.3 modified to do most work in memory.	5.73
xalancbmk	Transforms XML documents to other document types.	267.32
sjeng	A highly-ranked chess program that also plays several chess variants.	10.54

Table 1: Describes the basic characteristic of each SPEC CPU2006 benchmark.

There are four sources of overhead in our proposed technique. The first source is the overhead of the simulation, which is incurred by simply using Simics. This overhead can vary significantly. As shown in Table 2, the simulation times on Simics range from 6 to 33 times greater than the execution times of these benchmarks on a native system. Note that we set up our native system to have a similar configuration to that of the virtual platform. It has a 2.6 GHz Pentium 4 CPU with 512MB of memory. It also runs the same Fedora Core 5 Linux with the same 2.6.15 kernel.

Benchmark	Execution Time (seconds) Real System	Simics	Slowdown (times)
libquantum	5.98	144.79	24.21
perlbench	24.22	575.79	23.77
h264ref	49.21	850.77	17.29
gobmk	65.59	980.41	14.95
astar	27.88	296.88	10.65
mcf	21.11	181.36	8.59
hmmer	14.78	387.04	26.15
gcc	10.98	274.81	25.03
omnetpp	7.43	248.37	33.43
bzip2	57.11	347.65	6.09
xalancbmk	6.46	172.16	26.65
sjeng	17.91	339.70	18.98

Table 2: Simulation overheads of Simics.

The second 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 third source is SimTracer, which adds significant overhead to the process. According to Table 3, the overheads of SimTracer range from 3.5 to 28 times slower than execution times of Simics without it. Note that this overhead is in addition to the simulation overhead reported in Table 2. As such, the execution times of SimSight range from 27 times (bzip2) to 475 times (h265ref) slower than those of the native system. The sizes of generated log files range from 120KB to about 1GB.

Benchmark	log size (MB)	W/O SimSight (seconds)	W SimSight (seconds)	Slowdown (times)
libquantum	0.12	144.79	499.06	3.44
perlbench	4.1	575.59	2280.14	3.96
h264ref	65.4	850.77	23894.40	28.09
gobmk	1024	980.41	13799.93	14.07
astar	60.5	296.88	8215.06	27.67
mcf	7.2	181.36	2169.33	11.96
hmmer	13.6	387.04	2540.31	6.56
gcc	61.8	274.81	2301.20	8.37
omnetpp	45	223.27	2086.44	9.34
bzip2	8.3	347.65	1550.40	4.46
xalancbmk	23.8	172.16	743.58	4.32
sjeng	112.1	339.70	4358.4	12.83

Table 3: Tracing overhead for each benchmark and the generated size of each log file.

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 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 4 provides detailed distribution of the overhead based on these three runtime components.

As shown in the figure, 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's tracing module heavily contributes to the overhead. We also find that the logging component has very negligible effect on the overall overhead even when size of the compressed log is over 1GB. For the two applications with the greatest slow-down (*h264ref* and *astar*), the cost of processing function addresses dominates the overhead.

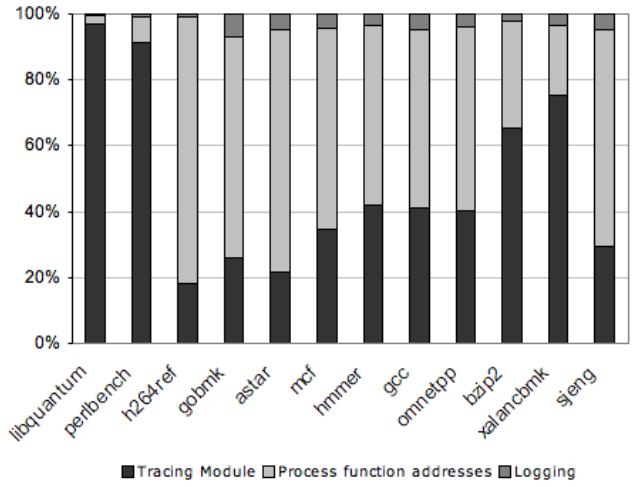

Figure 4: Overhead distribution of SimSight for each benchmark.

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 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. SimParser takes 61 seconds to parse the trace log of *libquantum* (less than 1MB of compressed information), 16 minutes to parse the log of *perlbench* (4 MB), 61 minutes to parse the log of *h264ref* (65MB), and nearly 3 hours to parse the log of *gobmk* (1GB). These long parsing times are mainly due to inefficient file I/O in Simics.

Running in a native system. The performance of SimParser is 10 to 73 times faster than 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.

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 in a native machine.

5. Usability Studies

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

5.1 Improved Program Understanding

Modern computer systems often employ advanced runtime systems to perform tasks that can make execution faster [19], overcome binary incompatibilities [4], ease the management of computing resources [20], and increase programmer productivity [10]. For example, a dynamic translator can be used to translate an architecture specific executable to run on another architecture or dynamically optimize an executable that may have been compiled for a previous processor architecture [8, 4]. Runtime systems such as Hardware Abstraction Layers [3] and garbage collectors [20] are used to simplify the management of hardware components and memory, accordingly. Dynamic loaders and linkers are used to simplify the task of managing shared binary objects such as libraries that are commonly used by applications [10, 23, 22, 1].

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 [20]. A dynamic loader and linker loads a dynamically linked object the first time it is accessed [1]. 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). As such, 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-2.4.so`) and any calls made by `ld.so` to its helper functions, allowing developers to observe when these runtime systems execute and determine module and function-call dependencies. Figure 5 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.

5.2 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. As such, maintaining library compatibility in systems that use dynamic link libraries is a challenging problem [10].

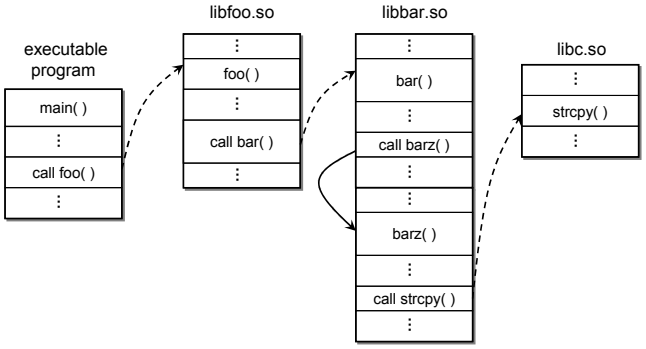

Figure 5: Dependency of our sample program

Figure 5 depicts a simple example that utilizes three shared libraries: `libfoo.so`, `libbar.so`, and `libc.so`. Note that a solid arrow represents a function call to a statically linked library. A dotted arrow represents a function call to a dynamically linked library. As shown in Figure 5, `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()`, 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, "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 shows the result from SimSight. In this case, the error is in function `strcmp` located inside `ld2-4.so`.

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

```
__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: Dynamic Call Graph of our sample program

6. 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 [42, 26, 24] then evaluate their effectiveness by running benchmark programs in the virtual platforms [35, 40].

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. [40] 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 [24]. 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 [2]. 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 [2, 40]; 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 should be acceptable to suffer significant runtime overhead in favor of greater visibility and completeness.

7. Future Work

We plan to experiment with on-demand tracing to reduce overhead and increase applicability. An example situation 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 such scenarios, a 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 [37, 30]. In the on-demand tracing mode, SimSight begins tracing from the snapshot location. By starting close to an execution point of interest, we can significantly reduce the tracing overhead by eliminating unnecessary tracing efforts while still generating meaningful interaction information. We will also explore an event-based technique to initiate tracing.

We are also working on extending SimSight to generate data-flow information. At the present time, we have built a prototype based on MicroC/OS-II running on MS DOS that can capture any data accesses to global variables. Our next step is to capture any accesses to variables on the stack. Additionally, we will extend SimSight to capture accesses to hardware devices (e.g., communication buses and I/O components) by a particular program. We imagine that such information can be useful in determining sources of resource sharing that may lead to contentions and some forms of concurrency errors such as priority inversions and deadlocks. Ultimately, we want to use SimSight as a tool to support testing and debugging of embedded systems.

Because the overhead to run SimSight can be quite large, efficient large-scale data management can play an important role in allowing the generated information to be quickly accessed by users. We are exploring an option to store the generated results in a database system so that developers can easily query the results.

8. 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. While our approach suffers from significant runtime overhead, we consider it acceptable since our tool is designed to be used during software development and not deployment. Furthermore, we believe that the overhead is a reasonable trade-off as virtual platforms can provide unparalleled observability at both the hardware and software layers.

We then implement a prototype in Simics full system simulator to support x86/Linux executables. To evaluate the overhead to capture function call information, we use 12 programs from SPEC CPU2006 benchmark suite. The result indicates that tracing function calls can make SimSight run 4 to 28 times slower than Simics with no tracing. As such, the execution times of SimSight are 27 to 475 times slower than those of a native machine. We also find that the parsing time is also not significant if it can be accomplished in a native system.

References

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

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

[3] Altera Corp. Nios Processors. http://www.altera.com/products/devices/nios/nio-index.html.

[4] Apple Inc. Apple Rosetta. http://www.apple.com/rosetta, 2007.

[5] A. Bahrami. Elf symbol tables. http://blogs.sun.com/ali/entry/inside_elf_symbol_tables.

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

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

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

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

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

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

[12] Eigenclass. Call graphs to analyze code dependencies, or just because. http://eigenclass.org/hiki/call+graphs.

[13] freshmeat. latrace. http://freshmeat.net/projects/latrace.

[14] Z. Frey. Coverage measurement and profiling. http://www.linuxjournal.com/article/6758.

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

[16] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–111, San Diego, California, USA, 2007.

[17] J. Haas. What is ltrace. http://linux.about.com/cs/linux101/g/ltrace.htm.

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

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

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

[21] U. S. Laboratories. Executable and linkable format.

[22] J. R. Levine. Linkers and loaders. http://www.iecc.com/linker/.

[23] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

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

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 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*, pages 190–200, New York, NY, USA, 2005. ACM.

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

[27] Microsoft. Microsoft portable executable and common object file format specification.

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

[29] G. V. Neville-Neil. Code spelunking redux. *ACM Queue*, 6(7):26–33, 2008.

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

[31] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 16–16, Washington, DC, 2003. USENIX Association.

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

[33] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *In Fundamental Approaches to Software Engineering, LNCS 3442*, pages 282–297, 2005.

[34] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–226, 1979.

[35] M. Schindewolf. *Analysis of CAche Misses Using Simics*. PhD thesis, University of Karlsruhe, Karlsruhe, Germany, 2007.

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

[37] B. Tuthill, K. Johnson, and T. Schultz. 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.

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

[39] J. Weidendorfe. Kcachegrind - call graph viewer. http://kcachegrind.sourceforge.net/html/Home.html.

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

[41] H. Yazarel, T. Kaga, and K. Butts. High-confidence powertrain control software development. In *SCDAC*, Nov. 2006.

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