DESIGN AND IMPLEMENTATION OF SIND, A DYNAMIC BINARY TRANSLATOR

by

TREK PALMER

B.S. Computer Science, University of New Mexico, 2001

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

December, 2003

DESIGN AND IMPLEMENTATION OF SIND, A DYNAMIC BINARY TRANSLATOR

by

TREK PALMER

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

December, 2003

DESIGN AND IMPLEMENTATION OF SIND, A DYNAMIC BINARY TRANSLATOR

by

TREK PALMER

B.S. Computer Science, University of New Mexico, 2001

M.S., Computer Science, University of New Mexico, 2003

Abstract

Recent work with dynamic optimization in platform independent, virtual machine based languages such as Java has sparked interest in the possibility of applying similar techniques to arbitrary compiled binary programs. Systems such as Dynamo, DAISY, and FX!32 exploit dynamic optimization techniques to improve performance of native or foreign architecture binaries. However, research in this area is complicated by the lack of openly licensed, freely available, and platform-independent experimental frameworks. SIND aims to fill this void by providing an easily-extensible and flexible framework for research and development of applications and techniques of binary translation. Current research focuses are dynamic optimization of running binaries and dynamic security augmentation and integrity assurance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program transformation and optimization are not new ideas; however the notion of performing them at runtime is a recent invention. The attraction of dynamic transformation and analysis of programs derives, in part, from the fact that the amount of static information available to a compiler is shrinking. Object-oriented languages that support dynamic loading and unloading of code impose a serious restriction on the ability of static analysis to effectively guide optimization. Even in a fairly static O-O language such as C++, there are considerably more challenges for a static compiler to overcome than in C. Static compilation techniques fail primarily because they have insufficient information to work with. When statically compiling a program with loadable modules, for instance, the compiler cannot make many assumptions about the internal structure of those modules, and consequently standard optimization techniques (such as inlining) become impossible. Also, for static analysis to be fully effective, it is often necessary to have access to the source code. For instance, some of the most effective security analysis programs (such as StackGaurd [4]) need full access to the source code to correctly protect vulnerable code segments. In reality, however, source code is often impossible to obtain.

1.1 Dynamic Binary Translation

Dynamic binary translation is a method to overcome these deficiencies in static compilation techniques. The basic idea is to dynamically monitor a running program and use the gathered profile (which contains a great deal of information) to guide further program transformations. The challenge is to do this efficiently and transparently. Both efficiency and transparency are difficult problems. To provide transparency, the binary translator must emulate all the idiosyncrasies of the underlying system, and do so in such a way that control never leaves the translator. This is the problem that most debuggers have to solve. However, whereas a debugger developer can choose to sacrifice speed for convenience and reliability, such trade-offs cannot be made with a dynamic binary translator. The efficiency constraint means that the system as a whole must be able to simulate the underlying architecture without significantly slowing down the running program. In practice, this means about a 10% slowdown is acceptable. This leads to the use of many arcane tricks and techniques to achieve the seemingly impossible goal of dynamic profiling with almost no slowdown. These are what makes designing and implementing a dynamic binary translator such an engineering challenge.

Most binary translators have the same basic components: something to profile the running code and gather traces, something to transform the traces into fragments, and something to link the fragments up with the program's address space and run the fragments directly on the processor. It is the last step that makes up for the cost of profiling and transforming running code. Because most of a program's execution is confined to a small portion of the code, if that portion is optimized and run directly on the processor it would speed up execution significantly. Identifying those hot sections of the code, however, is effectively impossible to do statically. Fortunately, the runtime characteristics of many programs are often much simpler than the static whole-program characteristics. This simplicity, along with the increased information available at runtime, makes it possible for dynamic translators to identify important segments of code that static analysis

would not be able to catch. This ability can be used to guide instance-tailored optimizations to improve program performance, or to guide runtime security transformations to improve program stability and security.

1.2 Why Another One?

While SIND is not the first attempt at a dynamic binary translator, it is the first open one. Many previous systems were created by companies as either an internal research tool[1] or as a commercial product [3], and consequently were never released. As the discussion in Chapter 2 reveals, many of these systems also have deficiencies or peculiar requirements. Some systems are so tied to the target platform that porting them was infeasible [1]. Other systems have specific (and unobtainable) hardware requirements [5]. SIND was designed with all this in mind. The SIND framework was intentionally constructed to be portable and the current implementation for the UltraSPARC can run on commodity hardware with no custom components. These advantages alone warrant the development of SIND. In addition, SIND is an open-source system and as such may become an important research tool with a large developer base. It was, in fact, the very lack of such an open tool that motivated the development of SIND.

1.3 SIND

SIND is my effort at designing a platform-independent dynamic binary translation framework, and implementing that framework for the UltraSPARC architecture (running Solaris/SPARC). This effort stems from the fact that there are no real open platforms for doing dynamic translation. This hinders research, especially in a field where open research tools are the norm. SIND's design was abstracted from several published dynamic binary translators and is aimed at providing a general framework for building a binary

translator for any given platform. The whole system is organized in an O-O fashion, with each major component as a separate module. This modularity is intended to aid initial and subsequent development. In the future, it should be possible to extend a module without having to modify any other code.

The current SIND system implements user-level integer instructions. No supervisor or floating point code is currently supported. The lack of supervisor code is not a problem because in a modern UNIX-like system (such as Solaris), all the supervisor code lives in kernel space. Requests to this code are made through syscalls, which are proxied by SIND. Floating-point code is less common than integer code, but for SIND to be truly useful it must handle floating point operations. These instructions were intentionally passed over due to the potential complexity of implementing them correctly (and the resultant debugging nightmare). SIND is also unaware of the Solaris threading infrastructure and may therefore be insufficiently thread-safe.

1.4 Overview of the Thesis

The remainder of this document is organized as follows: Chapter 2 gives a summary of the previous efforts at dynamic binary translation; Chapter 3 describes, in detail, the design of the SIND system; Chapter 4 is a discussion of the current implementation of SIND for the UltraSPARC architecture; Chapter 5 is a discussion and evaluation of the performance and flexibility of the current experimental system; Chapter 6 is an overview of the history of the project as well as a discussion of large scale technical issues encountered while implementing SIND; Chapter 7 describes how to use the developmental SIND tool; and lastly the document is concluded with an appendix describing the code itself and details such as directory structure.

Chapter 2

Previous Efforts

There have been several notable efforts at dynamic binary translation. The most well-known is the Dynamo project from HPLabs. This was a dynamic optimization research project for HPPA systems running HP-UX. Another interesting project was the FX!32 project from DEC (now part of HP/Compaq). This system was a dynamic binary translator that ran IA32 binaries on an Alpha (running Windows NT). FX!32 had several notable features: not only did it efficiently transform foreign binary instructions, it persistently stored the fragments on disk and optimized them in an offline batch-processing phase. Other interesting systems include the DAISY project from IBM, the Hotspot and Jalapeño/Jikes JVMs, and Transmeta's Crusoe 'code-morphing' technology.

2.1 Dynamo

The Dynamo system [1] is, in many ways, the seminal effort in this field. It is the most popularized effort that actually achieves noticeable improvements in running time. The Dynamo system is geared toward dynamic runtime optimization of HPPA binaries running under a custom version of HP-UX. The system is bootstrapped by a hacked version of

`crt.o` and begins running the binary immediately. The instructions are fully interpreted by a software interpreter, whose primary task is to identify and capture hot code traces from the running program. The profiling method used is one of the Dynamo team's fundamental contributions. They experimented with several profiling metrics and found that a simple statistical approach yielded the best combination of accuracy and speed. First, the profiler only focuses on code traces that started with *trace heads*, namely backwards-taken branches. These branches are indicative of loops within the program, and Dynamo assumes that this is where most of a program's work gets done. Secondly, the profiler assumes that, on average, the branches being taken when it examines the code would be the ones the program would normally take. Therefore when a trace head became hot (was visited enough times), only a single code trace would be gathered.

This code trace is then run through several simple compiler passes to yield an optimized fragment. Because overhead had to be small the compiler only performs simple, linear pass optimizations. The fragments are then loaded into the *fragment cache*. Dynamo's cache holds the other fundamental contribution. Rather than just linking the fragment so that it correctly accessed program data, the fragment is also potentially linked to other fragments already in the cache. This obviates the need to leave the fragment cache from one fragment merely to have to re-enter the cache to execute another fragment. This single improvement led to impressive performance increases.

Despite its many successes Dynamo has many disadvantages. First, the system is not open. This seriously hampers research, as the tool cannot be extended when necessary. Second, Dynamo is specifically tailored to the HPPA architecture and HP-UX operating system, to which I do not have access. Thirdly, Dynamo would be difficult to port, even if the source code was publicly available, the system wasn't engineered to be particularly extensible. The HP engineers who wrote Dynamo admitted that the whole system may have to be rewritten to be useful on another platform.

2.2 DynamoRIO

DynamoRIO [2] is the successor to Dynamo. It is also a closed, proprietary system, but it is designed for the Intel IA32 (x86) architecture and has versions that run under Windows and Linux. In addition to the standard problems of building a dynamic optimization system, DynamoRIO had to overcome the enormous cost of interpreting the dense and complex x86 instruction set. After several false starts, this was eventually achieved with the use of a so-called basic-block cache. This is a form of 'cut & paste' interpretation in which the interpreter/decoder fetches basic blocks from memory, rewrites branch/jump targets and executes the modified code directly on the processor. This alleviated the difficulty of actually interpreting x86 instructions, but made profiling more complex. The initial decision to use a basic-block cache also tied the rest of the system to the x86 architecture [1]. In the cause of efficiency, each component of DynamoRIO was written to be as specific to the x86 architecture as possible. As a consequence, the entire system is highly non-portable and would have to be completely rewritten to handle a new instruction set [Personal Discussions with Derek Bruening, the DynamoRIO Maintainer].

Despite the tight coupling between DynamoRIO and the x86 architecture, the system is more open and flexible than the original Dynamo. Even with the closed nature of the underlying source code, there is a useful API that allows outside developers to add to the system. However, such outside additions are restricted by the API and are slowed by the need to pass data through an additional interface not used by DynamoRIO internals. Despite the improvements over the original Dynamo, DynamoRIO failed to fully solve the portability and extensibility problems.

[1] The basic block cache works by rewriting branching instructions and executing the basic blocks directly on the processor. Therefore, a basic block cache is very architecture and OS specific.

2.3 FX!32

FX!32 [3] is a dynamic translation program from DEC. It is designed to translate IA32 binaries to Alpha code at runtime. The whole system runs on Windows NT for the Alpha, and existed because many NT developers were either unable or unwilling to write Alpha-friendly code. FX!32 has a number of notable features. It does a great job of translating foreign binaries, facilitated primarily by the fact that the NT API is standard across both the Alpha and IA32 platforms. This allows rapid translation of system and library calls in a 1-to-1 fashion. FX!32 also optimizes the translated traces, but in a novel way. Rather than doing optimizations at runtime the FX!32 system simply translates the trace and then saved the translated version to disk. Later on, a batch job examines the saved traces and optimizes them using potentially long-running algorithms. In practice, this means that each time a user ran an IA32 application it would be somewhat faster than the time before.

FX!32 is an interesting piece of software, but it too suffers from serious drawbacks. Primarily, it suffers from the fact that its a closed-source system. DEC (and later Compaq) sold FX!32 along with NT for Alpha, and considered releasing the code to be economically impossible. Also, FX!32 is closely tied to the NT platform, which can be difficult to develop for.

2.4 DAISY

DAISY [5] is a binary translation project from IBM that performs dynamic compilation of Power4 binaries. It is similar to Dynamo in principle, but it employs more sophisticated translation and profiling schemes. This allows DAISY to do a more sophisticated analysis than Dynamo. For instance, a limited form of control flow analysis across branches and calls is performed (to eliminate as much indirection as possible). However, this added power comes at the cost of larger runtime overhead. The DAISY project obviated this

cost by creating a custom daughter-board that would house an auxiliary processor to run the DAISY system. This secondary processor only has to run at a fraction of the speed of the main processor, and the daughter board has several megabytes of isolated memory available only to the auxiliary processor. Because of this, DAISY has automatic memory protection at no runtime cost, the additional hardware also removes the distinction between the operating system and user applications. This means that DAISY can optimize both OS code and application code (and even optimize call sequences from one through the other).

Unfortunately, the DAISY project never produced a commercially-available version of the DAISY processor in hardware. All the published results came from detailed software simulation of the proposed hardware. Even if the hardware were eventually mass-produced, it was intended for use in high-end servers, and so would probably have been very expensive.

2.5 Crusoe, JVMs, and Others

Other dynamic translation projects include the Code-morphing technology used in Transmeta's Crusoe processor [7], the HotSpot and Jalapeño/Jikes optimizing JIT JVMs, and other virtual machines that employ dynamic (otherwise known as Just In Time) compilation techniques. The main disadvantage with code-morphing is that in addition to being proprietary, it is specifically tied to the Crusoe VLIW architecture. The JVM and other language virtual machine projects, although useful from a design perspective, did not contribute much to the actual construction of SIND. This is due to the virtual machines being tailored to the needs of a specific language. This means that most language virtual machines, although closer to hardware than the uncompiled program, have features useful to the source language that are difficult to map directly to hardware.

Chapter 3

SIND Design

The SIND system design is not particularly revolutionary. It is a synthesis and extension of many dynamic translator designs. Because most dynamic binary translators have to solve similar problems, many have similar designs. This similarity is encouraging, because it means that if this structure can be expressed in code, the construction of new binary translators would be reduced to extending the base modules, rather than designing the whole system from scratch.

Figure 3.1 shows the major components of SIND. The interpreter is the module that handles the dynamic execution and profiling of the running binary. The transformers translate gathered traces into fragments. The fragment cache handles fragment linking and runs the fragment code on the processor. The memory and syscall-manager handle the system specific aspects of memory protection and operating system interaction, respectively. They are separated from the other modules to ensure as much platform independence as possible. Lastly, the bootstrapper and dispatcher initialize the other modules and handle inter-module communication.

This framework is generic enough to encompass all source and target architecture configurations, and separates the components so that they may act like ‘plug-in’ modules.

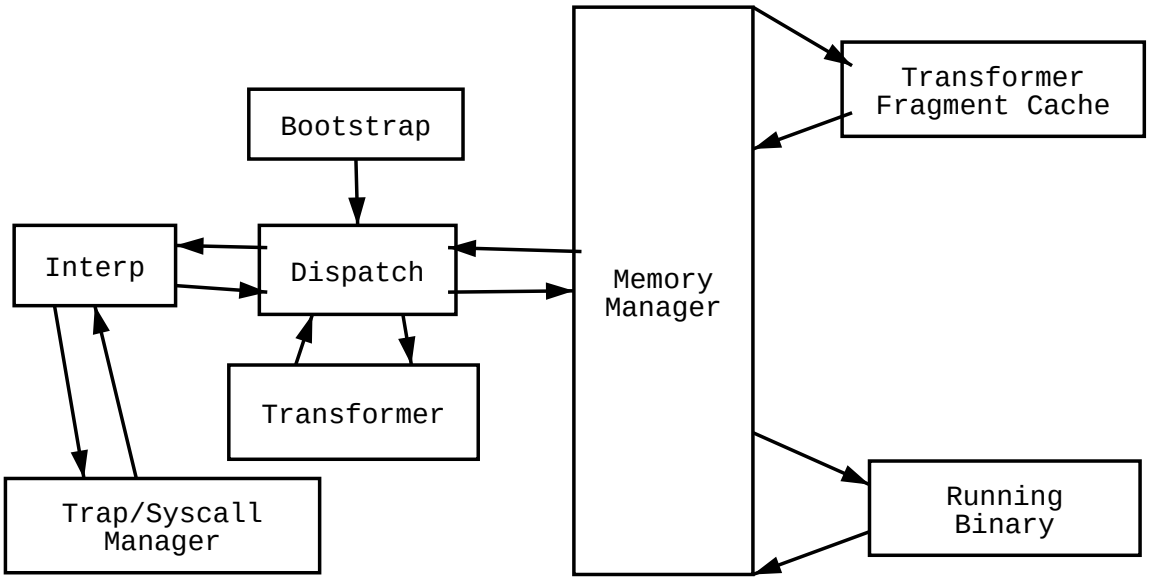

Figure 3.1: SIND modules

For instance, because the interpreter accesses memory through the Memory Manager and accesses OS functionality through the Syscall Manager, the interpreter only has to emulate the source architecture and has no dependence on the operating system. This means, ideally, that an UltraSPARC interpreter would be able to run (without modifying the interpreter source) on both an UltraSPARC and Power4 and would trust the Memory and Syscall Managers to take care of OS specifics. The intention is to isolate the interpreter from all but the most large-scale details of the target architecture. Basically, only the endianess and bit-width of the underlying system need to be taken into account (and this, only because C++ specifies no standards for the size and endianess of data).

The basic operation of the SIND framework is also platform-independent. The program to be run under dynamic translation is started by the bootstrapper. The bootstrapper assures that all dependencies (libraries and other shared objects) are loaded and halts execution just before the program starts. Then, control passes to the dispatcher, which initializes all the remaining SIND modules and starts up the interpreter. The interpreter

dynamically executes the program and gathers profiling information. According to some internal metric, the interpreter eventually decides that it has encountered an interesting code segment and gathers the relevant instructions and processor context into a trace. This trace is then handed (through dispatch) to the transformers. The transformers transform the trace into a functionally equivalent fragment. The nature of the transformations could be varied. Traces could be rewritten to be more efficient, but they could also be rewritten to be more secure, or to generate more fine-grained profiles. When the final transformer has finished its transformations, the fragment is handed to the fragment cache. The cache's primary responsibilities are to guarantee that the running fragment will have transparent access to all program data, and to simultaneously guarantee that the running fragment will not modify SIND or break out of SIND. The cache can protect SIND data by selectively write-protecting the regions of memory that SIND inhabits when the cache is entered and un-write-protecting the regions when the cache is exited. The cache also needs to check for system calls that might un-protect the SIND memory regions. It can do this by placing itself between the executing fragment and the eventual system call, and checking on the parameters passed in by the fragment. The cache can guarantee that an executing fragment will be able to access all program data by performing a final rewriting of the fragment in a process analogous to dynamic linking and loading. From then on, when program control reaches the address of a fragment, control is passed to the cache, and the fragment then runs directly on the processor. When control leaves the fragments in the cache, the SIND interpreter starts up again and continues dynamic program execution.

3.1 Interpreter

The interpreter's main function is to gather profiling information and code execution traces. These are passed to transformers, which use the profiling information to guide specialized transformations of the code traces. Because one of the goals of the SIND sys-

tem is to do runtime binary optimization, it is vital that the interpreter should introduce as little overhead as possible. As a consequence, the interpreter must be very efficient and every reasonable effort must be made to improve its speed.

The first interpreter to be fully designed and implemented in SIND emulates the 64-bit SPARC v9 architecture. The design was motivated by several factors: first, because SIND runs in non-privileged mode, the interpreter is primarily a non-privileged instruction interpreter; second, the interpreter only needs to be functionally correct, therefore no complicated hardware structure needs to be emulated in order to produce accurate simulation. The interpreter's job is then to replicate a user's view of the processor and discard any lower-level structure that interferes with the efficient execution of code.

3.1.1 Registers

The interpreter replicates user-visible registers as an array of 64-bit quantities in memory. On a 64-bit host machine these are native unsigned 64-bit integers; on 32-bit machines they are two-element `structs`. There are several caveats, however. The SPARC architecture supports register windows for integer registers. This was emulated by allocating a large array of 64-bit quantities, setting the lowest 8 to be the global registers, and having a window of 24 registers slide up and down the array as procedure calls are made and registers are saved and restored. It is important to be able to restore the user stack in order to be able fully to emulate a system call. It is also important to keep SIND's own stack separate from the user stack, because the interpreter runs in the address space of the user process and so in principle the user process's stack entries might clobber the interpreter's stack. Creating and maintaining two separate stacks is discussed below, but the discussion of restoring the user stack is in the Syscall Manager section.

Maintaining two separate execution stacks requires a bit of system hacking. The last valid stack frame is left alone, and its stack pointer (pointer to the top of the frame) is saved

for reference. A new page is allocated for the separate stack, and its topmost address is recorded. This topmost address is to become the new frame pointer. Then an explicit `save` instruction is issued; it creates a new register window, but with the stack pointer pointing into the new page. Then the frame pointer register can be manually set. From that moment on, all further calls should write their stack data to the alternate stack page(s). Apart from protecting the SIND call stack from manipulation by the interpreted program, this also means that SIND's stack can be `mprotect`-ed to safeguard its contents when executing code directly on the processor (either issuing traps or when in the fragment cache).

The floating point registers on the SPARC consist of three overlapping sets of 32, 64, and 128-bit floating point registers. There are 32 32-bit, 32 64-bit registers, and 16 128-bit registers. The 128-bit and 64-bit registers overlap completely (e.g., the first 128-bit register is the same as the first two 64-bit registers), and the 32-bit registers overlap with the bottom half of the other two. This was implemented as a contiguous region of memory, accessed in different ways depending upon the instruction used (some checking had to be done to make sure no accesses were attempted to non-existent 32-bit registers).

3.1.2 Instructions

Although the SPARC v9 architecture is 64-bit, the instructions are still 32-bit, which allows backward compatibility (consequently, the software interpreter is also capable of running SPARC v8 code). The SPARC has 30 different instruction formats, grouped together into 4 major families. However, these formats are all the same length (32 bits) and were designed to be quickly parsed by hardware. This permits streamlining the fetch and decode portions of the interpreter. Each instruction format was specified with its own bit-packed struct, and all such structs were grouped together in a union with a normal unsigned 32-bit integer. Each format family is distinguished from the others by the two high-order bits of

the instruction.[1] Thus the interpreter has jump tables for each instruction format family (actually three jump tables and one explicit function call), that are keyed by the opcode, whose position depends upon the format family. A case statement branches on the two most significant bits to the correct jump table, and then the correct function is called.

3.1.3 Exceptional Conditions

Occasionally during execution, an instruction will cause an error. The SPARC v9 architecture manual clearly defines these exceptions, and, for each instruction, specifies which exceptions it can raise. Many of the exceptions are caught by the operating system and used to handle things like page faults and memory errors. Non-recoverable exceptions usually cause the operating system to send a signal to the executing process. To mimic this, if the interpreter decides a given instruction would cause an exception (such as divide-by-zero), then a procedure similar to that used for system calls can be used. The running binary's state is restored on the stack, and then the interpreter executes the offending instruction directly on the processor. This generates the appropriate operating system action (usually, killing the process).

3.1.4 Signals and Asynchronous I/O

In the Solaris operating system there are really only two ways of communication between user and supervisor (kernel) code. One, the system call or trap, is discussed in the Syscall Manager section. The other, signals, had to be dealt with differently. Because the SIND system is guaranteed to be loaded before all other libraries, its definitions of functions will take priority (if they're exported). The SIND interpreter interposes on the signal

[1] To be precise, Format 3 and Format 4 both can have the values 10 or 11 in their upper bits, but in SIND Format 3 instructions are instructions with 10 in the upper bits and Format 4 instructions have 11 in the upper bits.

functions, and registers a special handler for all registrable signals. Thereafter, when the interpreted program registers a signal, it will go through SIND's registration system, rather than the system's. This means that SIND has to record the signal handlers registered by the program (in order to execute them when a signal is generated). When the OS sends the process a signal, it will be first intercepted by SIND, which will need to start interpreting the handler registered for that signal. In this way a signal cannot cause control to leave the SIND system.

3.2 Memory Manager

The SIND memory manager provides a generic interface between the process's address space and SIND's (possibly separate) address space. In the non-architecture specific memory manager, the interface consists of a small number of memory access and modification functions. To access memory, there are `ReadByte`, `ReadHalf`, `ReadWord`, `ReadDoubleWord`, and `ReadQuadWord`. These functions take an Address argument (the size of which is determined at compile time), and return the data located at that location. The names are meant to convey the size of value returned and follow the modern RISC convention of a word as a 32-bit value. But this does not mean that somehow the memory manager interface is only appropriate for RISC machines. To modify memory there are corresponding `Write ...` methods.

The interface exists as a separate module to support variations of SIND in which the interpreter and transformer exist in separate address spaces. In fact, the initial SIND system did exist in a separate space and accessed the processes address space using `procfs` [9]. For the current incarnation of SIND (which inhabits the process's address space), the memory manager just checks the address (to make sure that the program is not modifying SIND), and dereferences it appropriately. The `Read` and `Write` methods are also prefixed with `inline` directives to further eliminate overhead.

3.3 Syscall Manager

Because SIND sits on top of the kernel and only interprets user-space code, system calls must be executed directly on the processor to initiate the correct kernel action. SIND cannot simply execute the trap instruction directly, however the interpreted process's state must be reincarnated in the hardware, and then the trap can be issued correctly (returning into SIND, of course).

To restore the user stack, the original stack top (before control was passed to SIND) address must be preserved. The simulated stack (in the register windows array), must then be copied over to the stack area before the system call can be made. However, just copying the registers is insufficient. Each stack frame may have an arbitrarily large spill area, and that must also be preserved and copied over for trap emulation. Each time a save instruction is issued, it is remembered so that the stack offset for each frame can be properly reconstructed. However, the spilled variables do not have to be remembered. Because the interpreter is executing in the same address space as the target process, values written to the spill area will be at the correct location for the stack, so the stack frame and its corresponding registers just need to be copied around such spilled variables. However, even this is not enough to completely recreate the user stack. The register window state must also be replicated in the underlying processor. Basically, this means 'rolling back' the current stack to its state when SIND took over and then pushing on all the necessary frames. When rolling back the stack, it is necessary to save the stack frames as they are deallocated (because they will need to be restored before normal execution can resume). In practice, because the two stacks are kept separate, this means `mprotect`-ing the interpreter's stack area and issuing the necessary number of `restore` instructions. When the stack has been rolled back to its starting position, the simulated register windows need to be copied to the processor and then explicitly saved to the stack. Although this is also time-consuming, we get register saving around spilled variables automatically.

3.4 Trace Gathering

The identification and collection of traces happens within the interpreter but can be considered a separate subsystem. This is possible because the trace identification and gathering code is completely independent of the rest of the interpreter (but for efficiency reasons is part of the interpreter object). Identification of traces happens at the instruction level; an instruction with the potential to be interesting is identified as it is being emulated. On the side, the trace gathering system maintains a data structure (currently a hash table) that contains all encountered trace heads. When an interesting instruction is being emulated, the tracing code checks to see if it has been encountered before, increments its counter if it has, and inserts a new record if it hasn't. In the current experimental system, the trace gathering code looks for branches whose target is behind them (a characteristic signature of a loop). In order to avoid gathering traces for rarely executed code a certain number of iterations have to pass before a trace head can be considered hot. Currently, the system also has a threshold of 15, which means that on the 15th execution of the same potential trace head the instruction is assumed to be a genuine trace head and trace gathering can begin in earnest.

A trace is simply a sequence of instructions gathered after encountering a trace head. There are no restrictions placed on the termination conditions for a trace, however the current system will stop gathering instructions if a certain numerical limit is reached (currently 256), or if the trace head is encountered again (indicating that we have completed one iteration of the loop). Note that these restrictions do not prevent a trace from including a complete function call (and the corresponding function's body), so function inlining is essentially free. A trace is essentially an array of instructions, each of which may have some annotation (for instance, it might be useful to record whether or not a branch was taken). A trace also includes an array of registers that contain the machine state when the trace head was encountered. This information is then passed on to the Dispatcher for subsequent transformation and insertion into the fragment cache. If these operations com-

plete successfully, the entry in the trace-head data structure is updated, so that next time the interpreter will jump directly to the fragment cache, rather than interpreting the trace.

3.5 Transformers

A SIND transformer has a simple interface: it accepts a trace and returns a trace. In this way transformers can not only optimize a particular machine code trace, but can be used to convert a trace from one machine to another, or to convert a trace from machine-level to an intermediate form more appropriate for optimization. Input validation is accomplished by extending the base trace class to a concrete, platform-specific version, and writing the transformers to use the most specific trace class. Then, if an incorrect type of trace is handed to a transformer, the type error will be caught during compilation. The proper sequencing of Transformers is the job of the dispatcher and is described in detail later.

3.6 Fragment Cache

The fragment cache has a simple interface. Apart from constructors, the fragment cache has only two real methods; the first takes a new trace and inserts it into the cache, and the second executes a fragment (keyed by the program counter) stored in the cache and returns the new processor state (for the interpreter). Internally, however, these functions are far from simple. The fragment cache maintains three sets of data. The first is the fragments themselves, the second is the fragment prologues, and the third is the fragment epilogues. The prologues act as guard code to fragment entry. They perform any checks specific to a fragment. These checks may be dictated by the types of transformations applied (for instance, constant folded instructions may need to check and make sure the constant hasn't been changed). The epilogue serves as the fragment's only exit point. All possible exit points (such as branches, `call`s, and `jmp`s) have their target addresses changed so that

on exit from the fragment code, they jump to the epilogue. The epilogue's job is then to clean up after the fragment, capture the processor's state, and then jump to the fragment cache's function to return control to dispatch.

The insertion function (`newTrace()`) takes a fragment, generates a prologue and an epilogue, and rewrites crucial instructions to correctly jump to the epilogue. This function then has a reduced instruction parser that looks for these crucial instructions, rewriting them as it copies them from the trace into the cache. The execution function (`jumpToCache()`) has a simpler job: it jumps directly to the prologue code for the appropriate fragment. After the fragment has finished, `jumpToCache()` packages up all the new processor context in a class wrapper (`FragExitContext`) and returns to the dispatcher, which updates the interpreter's state accordingly.

3.7 Bootstrapper and Dispatcher

3.7.1 Bootstrapper

The SIND bootstrapper has the task of correctly halting normal execution, initializing the SIND modules, and then transferring control to the interpreter. The current incarnation of SIND for Solaris/SPARC is a preloaded library that loads itself before any other libraries (excluding ld.so). The bootstrapper runs as the `.init` function for this preloaded library. The bootstrapper first initializes the SIND modules in memory, then sets SIND's signal handler to handle all handleable signals, and finally overwrites the first two instructions from the `_start` symbol with a call to the `SINDstartup()` function. Therefore, when the bootstrapper initialization function has finished, all signals generated during normal program execution will be correctly intercepted by SIND and when all the initialization routines of all the dependencies are done, control will return to the bootstrapper.

The `SINDstartup()` function's job is to correctly capture the state of the process at the `_start` symbol and then jump into the interpreter's `executeLoop()` function, which does the actual interpretation. The process's state is captured with a bit of SPARC slight-of-hand. The address of the interpreter's registers is loaded into a register known to be zero, and then each register is stored into the interpreter's register array at the correct offset. `SINDstartup` then allocates several pages for SIND's separate stack, initiates the new stack with an explicit `save`, and overwrites the new framepointer (passed through global registers). Thus `SINDstartup` forms a buffer between the process and SIND itself, but because `SINDstartup` has no immediate variables of consequence (everything is stored in the interpreter object), the user process can overwrite its stack frame without adversely affecting SIND.

3.7.2 Dispatcher

The Dispatcher's main purpose is to serve as a common interface through which all the SIND modules can interact. In particular the dispatcher coordinates all the details of trace processing. The transformers are akin to optimizations in an optimizing compiler and it is the responsibility of the dispatcher to properly sequence the transformers (as well as handle any shared data needed by several transformers). This ordering is very specific to the transformers used, and is an additional detail not necessary to the functioning of either the interpreter or the fragment cache. Once all the transformers are done, the dispatcher then hands the annotated machine instructions to the fragment cache. If, for instance, the transformers operate on a different instruction set (either an intermediate representation or a foreign architecture), the dispatcher will need to further transform the code to the architecture that the fragment cache wants.

Chapter 4

SIND Implementation

4.1 Overview

SIND is currently implemented in a subset of C++ that is basically C, but with classes rather than structs. This exploits the convenience of C++'s object-oriented infrastructure, but avoids the more troublesome aspects of the language, such as templates and iostream. I/O is done using the traditional C functions, but dynamic memory is allocated with the `new` operator. The bulk of the SIND system is implemented as classes (the only exceptions being the bootstrapper and the signal handling code). The classes form interface-implementation pairs, where an implementation inherits from its parent implementation and implements an interface that inherits from its parent's interface. Although this is technically multiple inheritance, only one non-interface class is inherited from so all the complications endemic to C++-style multiple inheritance are avoided. The following figure 4.1 illustrates this relationship in terms of the abstract `CPU` class and the concrete `SPARCCPU` class.

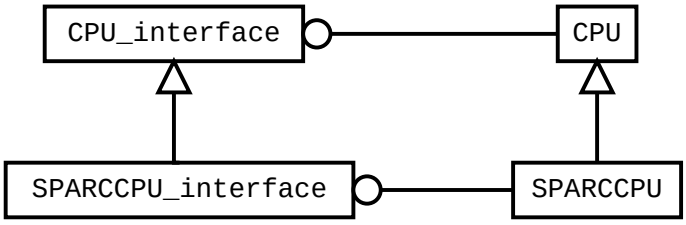

Figure 4.1: `CPU` inheritance tree

4.2 Interpreter

The SPARC interpreter is implemented in the class `SPARCCPU`. Internally, the class contains the array of registers needed to implement register windows (as described in subsection 3.1.1). `SPARCCPU` also contains representations of all user-visible registers (like the condition code registers). The only other significant amount of internal data to the interpreter are the jump tables for instruction decoding/execution. There are 4 format classes of instructions in SPARC, and each one is keyed by the 2 high order bits in an instruction word. Format 1 has only 1 instruction (the `call` instruction) and so when decoded, the method implementing `call` can be called directly. All other formats need further decoding. However, SPARC is RISC, so this decoding is very simple. Each format family has an additional op specifier (for instance Format 2 has an `op2` field), and the specific instruction is keyed by this op field. Therefore the methods implementing instructions are grouped by format into jump tables, and a switch (on the format number) will then call the function stored at the op specifiers offset in the appropriate jump table.

All of the instruction implementation methods return an `int`. Following the standard C programming style, a return value of 0 indicates success. Non-zero returns indicate failure. A positive result indicates a SPARC exception (such as *divide-by-zero*). These need to be emulated correctly (as described in subsection 3.1.3). Negative return values indicate a SIND internal error (such as an unimplemented instruction, or strange stack usage). These will usually cause the interpreter to spit out a error message and exit. The

error conditions are defined in the `SPARCExceptions.h` file.

In general, the implementation of the interpreter was a straightforward if time-consuming task. Certain instructions were non-trivial, however, and a source of constant frustration. `save` and `restore` are two particularly notable members of this category. Their implementation was complicated by all the array manipulation they had to perform. Off-by-one errors, or incorrect specification of boundary conditions were a particular problem.

4.3 Bootstrapper

The bootstrapper is perhaps one of the ugliest and strangest pieces of C code I have had to deal with. The current version is actually the third iteration of bootstrappers for SIND. It exploits the `LD_PRELOAD` functionality to load itself into the process's address space, and overwrites the first two instructions to jump to SIND. Getting this to work involved a lot of research into the Solaris linking infrastructure [8] [6] [11]. The bootstrapper has to locate the `_start` symbol, and then overwrite part of its code (which involves some mucking about with `mprotect`). The implementation of the bootstrapper was plagued by all the problems associated with system code. Poor documentation leading to poor code, strange documentation leading to strange code, and of course, the ever-present threat of undocumented features. The bootstrapper's greatest nemesis was in the form of such undocumented behavior. To locate `_start` the bootstrapper uses the `dlsym()` function to find the address. According to the documentation for the `dlfcn` functions, `dlsym()` needs a handle to the loaded object in order to work correctly. Such a handle is provided by a call to the `dlopen` function. Normally, `dlopen()` is used to open dynamic objects on the fly (as Apache does with its loadable modules), but if handed a NULL rather than a filename, `dlopen()` will return a handle to the current object. This is the method recommended in the documentation, yet when `dlsym()` was handed this handle it returned a strangely offset value of `_start` (it was, in fact, always 200 bytes away from the real

location). This odd behavior led to numerous errors, and was finally dispelled by consulting independent code examples (that is, code not written especially for the Solaris linkers guide[11]). This is a problem known to a small group of people on line, and has something to do with accessing symbols that begin with a "_". The solution is to use another form of `dlsym()`, and hand it a NULL handle. This will default to the current object.

4.4 Fragment Cache

The fragment cache was another tricky piece of coding. Although the SPARC instruction set is RISC, control transfer instructions are scattered throughout all the format families, so the mini-decoder could not be compact. As the instruction trace is stepped through, each instruction must be checked to see if it can leave the fragment. If it can, its target address (or the code following it, if not taken) must be adjusted to jump into a part of the epilogue. This part of the epilogue is then generated which will remember the PC of the exiting instruction and then transfer control to the state capture/cleanup code. Because the current fragment cache doesn't do any fancy internal linking, the code is not particularly complex, just rife with detail. Needless to say, this code is the resting place of many bit level errors and other demons of the assembly/machine code world.

Chapter 5

Evaluation of SIND

5.1 Performance of SIND

To measure the performance of SIND, several sample programs were run, and their execution times compared against those when using SIND. The programs in question were the simple test programs used to debug SIND itself. These programs were used rather than conventional benchmarks because SIND cannot currently run SPECint to completion.

5.1.1 Speed of Interpretation

The interpreter's speed was measured by timing the main loop of the `executeLoop()` function in the interpreter. The timing was performed using the high-resolution timing facilities of Solaris. Trace-gathering and fragment caching were both turned off, so the execution was completely within the interpreter. Debugging statements were also disabled[1].

As the table shows, the emulator introduces a slowdown factor of 150 to 225. This

[1]Initially, I erroneously gathered data with them enabled first, which allowed me to see that SIND's voluminous debugging output causes an additional slowdown of roughly a factor of 20.

test	description	slowdown
sind_test	arithmetic and logic test	140x
sindIO_test	hello world	210x
sindTimingTest	timing calls and I/O	223x

Table 5.1: SIND interpreter slowdown

multiple is consistent with an inspection of the compiled code for the SIND interpreter. The simple arithmetic functions have little or no branching statements and are between 100 and 150 instructions in length. The more complicated instruction functions, such as those for branching, are many thousands of instructions long, although most of that code lies in mutually exclusive branch targets. Inspecting a trace of instructions running from the initial decoding, through condition code checking, etc., shows instruction counts of between 200 and 250. Traps are the longest running instruction. The `tcc` SPARC instruction has to do all the condition code checking of a normal branch, but control is passed to the `handleTrap()` function, which must restore program state to the processor and execute the trap directly on the hardware. The situation is even worse for a syscall (`ta 8`), because it must pass through the `handleSysCall()` function first, which checks the arguments to make sure that the syscall will not side effect SIND itself. Fortunately, traps are rare.

Most of the interpreter's time was spent in the linking code. A program such as hello world, will require the execution of roughly 19,000 instructions [2]. Most of the time is spent in the linker back-patching the procedure linkage table with dynamic function addresses.

[2] Specifically, 19177 instructions.

5.1.2 Speed of Cached Fragments

Because the current system has no translators of note, the fragments in the cache are not very different from the gathered traces. The branch and jump targets have been rewritten, but no instructions have been eliminated. As a result, the fragment executes at almost the same speed as the native code. The prologue only adds 2 instructions (currently), although future transformers could enlarge this. Although the epilogue is many instructions long, it is not executed in its entirety. On an exit from a fragment, only two instructions would be executed. The first instruction is to jump to the fragment cache exit code, and the second loads the exiting PC value to a memory location (this instruction is in the branch delay slot). Therefore, the fragment is only 4 instructions longer than the original trace.

5.2 Memory Footprint of SIND

Almost all of SIND's data is statically allocated before hand or is on the stack. Those data structures that use dynamic memory have a default constructor that statically allocates a fixed amount. SIND can be built so that it allocates no dynamic memory. In this case, SIND occupies 440K of space [3]. This footprint will grow, however, as SIND is made self-contained (as explained in 7.2).

5.3 The Agility of SIND

The current experimental SIND system is capable of running 'toy' programs such as `hello world`. Current development efforts are focused on getting SIND to execute the SPECint2000 benchmarks (with the exception of the mtrt benchmark which is multi-threaded). Because the toy programs generate only a few long running loops, the fragment

[3] Actually 451568 bytes.

cache is insufficiently tested by them. Therefore, the bugs in the current system that prevent execution of SPECint are almost certainly in the fragment cache.

Chapter 6

SIND: A History

6.1 A Brief History

SIND began its life as a class project in a Java seminar (Spring 2001). Dino Dai Zovi and I were inspired by the Dynamo paper to try and implement a similar system for the SPARC. Very shortly we discovered why it took a small team of HP engineers over a year to construct Dynamo. We began with the ISEM [1] source, but soon ran into two limitations. First, ISEM only interprets 32-bit SPARC v8 code, and second (and more importantly) ISEM is a whole system emulator. ISEM emulated both privileged and non-privileged instructions, as well as a fully-featured memory subsystem and S-bus-style system bus. This was too much overhead to deal with, and after a summer of sporadic hacking, I decided that it would take less time to code an UltraSPARC emulator from scratch than to rip out the salient parts of ISEM. Also, initially we used `ptrace` [10] to access the running process, and had lifted most of our (poorly understood) bootstrap code from gdb. This led to a whole stream of difficulties with `ld.so` and memory protection. By the Christmas of 2001, I had given up on the ISEM `ptrace` combination and started coding the SPARC v9

[1] The Instructional SPARC EMulator, a 32-bit SPARC v8 emulator written by Barney Maccabe

interpreter. Dino Dai Zovi elected to start a PowerPC interpreter, but because of conflicts with work, school, and other research obligations could only make infrequent and minor contributions to SIND.

By the following summer I was the only developer working on SIND, and was mostly finished with the core interpreter. By the fall, the interpreter was sufficiently complete to warrant construction of a new bootstrapper. This lead to several branches of experimentation. Initially, I attempted to use the Solaris `procfs` facilities, but this proved to be overly complex and fraught with peril. I then decided that in the interest of both efficiency and ease of coding, I would locate SIND in the same address space as the running program. There were several implementation options: I could modify a system program (such as `ld.so` or `libcrt.o`) to automatically load SIND whenever a program was loaded; or I could use the `LD_PRELOAD` trick to cause SIND to be loaded as a shared object. The former option had the disadvantage of making SIND difficult to install and maintain (after all, I would have had to keep up with the latest release of whatever system program I modified), and the latter option had the disadvantage that my code would have to be massaged into position-independent library code. The latter option was more general and ultimately easier however, and so, compelled by my innate laziness, I made SIND a preloadable library.

The bootstrapper itself was the scene of some vicious entanglements with the Solaris Operating Environment[6]. I initially caused execution to return to SIND code by `mprotect`-ing the `_start` symbol's page to non-executable. Thus, when control finally transferred to the target program, a segfault would be triggered and intercepted by my handy bootstrapper cum signal handler. This had the disadvantage of being inexact and limited by the restrictions on actions inside signal handlers. This approach was abandoned by Spring of 2003, and replaced with the current bootstrapping system. The new system is exact and considerably more complete than that which it replaced. The spring also saw the stabilization of the interpreter, the beginnings of a trace-gathering system, and the

introduction of the Syscall Manager.

6.2 Experiences from the Design and Implementation of SIND

The design of the interpreter itself presented several challenges. There are two main options for an instruction set interpreter. It can be a full-fledged software interpreter, emulating the source instructions in software, or, if we are planning on running it on the same architecture as the instructions, we can do a 'cut-and-paste' interpreter. The cut-and-paste solution (otherwise known as a basic block cache) works by copying each instruction encountered to an area in memory, remembering to rewrite control-transfer instructions to jump to the correct new locations and then executing these copied instructions directly on the processor. This is a very lightweight interpretation system, and because it requires a decoder only capable of distinguishing control transfer instructions from the rest, it is the preferred solution on x86 platforms (systems such as Valgrind [12] and DynamoRIO [2]). However, such cut-and-paste systems have one major disadvantage. They can only be run on the platform whose instructions they are interpreting. This presented a disadvantage for our work, because we not only wanted to explore dynamic optimization, but also foreign binary execution (a lá FX!32 [3]). If we wanted to run this interpreter on another platform, it would have to be a full-fledged instruction set emulator.

The core interpreter itself is not complicated: emulating a compact RISC machine is not too difficult. Most of the effort went into the bootstrapper and system call subsystems. The bootstrapper itself has gone through many permutations. In the end, there were two major options. Either the interpreter starts itself up in the library initialization routine, or it causes control to transfer from the target binary's `_start` symbol into the interpreter. The problem with the first option is that it halts the loading process halfway through.

Normally the binary and all its dependencies are loaded and then, in the loading order, all the dependencies have their initialization routines called. If SIND were to take over in its initialization routine, then it would have to act like the loader and finish process loading. The second option, though it appeared to be more complicated, actually turned out to be the easier route. In SIND's initialization routine, the bootstrapper `mprotect`s the loaded `.text` segment to allow writes. The first two words/instructions after `_start` are saved to a reserve area, and then are overwritten with an explicit call instruction into the interpreter's code. This means that SIND will only be started *after* the loader has finished. This system is imperfect, however. If any loaded library prevents control from transferring to the start symbol (such as, for instance, by never exiting the initialization routine), then SIND will never be entered. This is not a big problem, however; because the SIND bootstrapper can easily be replaced without affecting the interpreter, a more thorough system can be developed and inserted without difficulty.

The syscall subsystem was discussed thoroughly in the design section above and took time to develop simply because of its complexity (almost all owing to the use of register windows). Development on the whole system was hampered by several tool deficiencies. The debugger we were using (gdb) has only limited support for 64-bit objects, and this severely hampered diagnosis. The debugger was also of limited use because we were not, in fact, debugging the running program: we were debugging a library that was loaded with the Unix `LD_PRELOAD` facility. Trying to use gdb's built-in facilities turned out to be more trouble than it was worth. The best method we discovered was to compile SIND with debugging on (and explicit stabs support), and cause an intentional segfault (by dereferencing `NULL`) near the suspect method. We could then load the core into gdb, and it would often give us enough information to help with debugging. When we needed more control, we inserted an `__asm__` block with an explicit debugger trap (`ta 5` on Solaris/SPARC).

Debugging, in fact, has proved to be the most complicated part of developing SIND. In an effort to make debugging easier I developed two tools. Both were actually modifications

to SIND in order to capture state correctly. The first was the driver. This was a program that stood between SIND and the running program and hand-fed SIND an instruction at a time. This was useful for debugging individual instructions in the interpreter, but not at all useful for dealing with all the strange interactions possible when SIND was in the process's address space. When debugging SIND in the process's address space, simply dereferencing NULL to cause a segfault was insufficient. `gdb` would not correctly load the preloaded library and so certain things (like `%sp` and the PLT) would be different than when the program crashed. In order to get a clear picture of memory at the point of interest, a 'BOGUS' flag was added to the SIND build process. When compiled with BOGUS, the SIND binary would insert itself into the running program's address space, and trigger a transfer of control at `_start`. However, control would not enter the interpreter, instead the overwritten instructions would be replaced and the PC of interest would be overwritten with a `jmp` to the BOGUS function that would print out the relevant parts of memory and then quit. These values would be those computed by the binary on the processor itself, and could then be compared against the voluminous debugging output of the normal (that is, not BOGUS) SIND.

The GCC compiler itself introduced problems. We were originally using the gcc 2.95 compiler collection which had somewhat buggy 64-bit support. The biggest problems were with C++ name mangling. In older versions of gcc, symbols defined in .c files or header files whose implementations were in .c files used normal C linking. That is, a function defined as `void foo()` was exported as the symbol `foo`. In gcc3 and up, anything touched by a C++ file was made to use C++ linking. C++ linking involves a technique known as *name-mangling*, whereby the symbol name has characters appended or prepended to it that the system uses to extract type information. Therefore a function `foo` in a class `bar` gets mangled to something like `_ZNKbar1fooEv`. This meant that many of the function interpositions we had created were no longer working when we upgraded to gcc 3.2 because their symbolic names were mangled beyond recognition. The way around this was to devise macros to enclose C-style definitions in a way that tells the

C++ compiler to leave them alone.

The GCC compiler also caused problems with register usage. On Solaris/SPARC systems, a shared object cannot write to the `%g2` or `%g3` registers (which are dedicated to passing values to syscalls). With gcc it is simple enough to specify not to use either global register; however, it is not possible to tell it to only avoid *writes* to those registers. This means that any code we have that explicitly copies values from `%g2` or `%g3` has to be compiled separately and then linked in later, which is cumbersome. On a load-store architecture, it should be trivial for an assembler to determine whether an expression that references a register is writing to it or just reading it!

Chapter 7

Using SIND

7.1 Invoking SIND

Because SIND is force-loaded into the running binary's address space, SIND must be a dynamic object. When SIND is finished building, there will be a file named `libsind.so` in the directory. To use this, a shell script is provided (`run_sind`). This will set `LD_PRELOAD` correctly and invoke the supplied binary. For example to run the hello_world program, one would simply type '`run_sind hello_world`'. There are some caveats, however. Because SIND uses `LD_PRELOAD`, there are some restrictions. To prevent an attacker from installing a malicious library in their home directory and causing everyone to interpose with their dangerous code, Solaris (and other ELF OSs) requires that any setuid program can only load `LD_PRELOAD`ed libraries from certain 'safe' areas. Under Solaris this can be configured using the `crle` program.

7.2 Extending SIND

The current experimental SIND system is not functional enough to be of much use to the average user, but because of its progressive open source licensing, SIND can be readily extended by needy coders. At the moment SIND will only work on 64-bit Solaris/SPARC system. An effort has been made to make the system capable of running on a 32-bit platform, but that code has not been tested and is certain to contain numerous bugs. SIND has also not been rendered self-contained, which is to say that I/O functions and memory management have not been replaced with local, specialized functions. For programs that 'play nice' this isn't a big problem. But for any advanced application (that may define its own `new` operator, for instance) SIND may fail horribly. Therefore SIND's I/O and memory needs must be met within SIND. This should not be a particularly difficult task, however. Many of SIND's data structures are fixed size and so dynamic memory is rarely used. In addition, when debugging is disabled, SIND performs no file I/O, and contains only a few print statements. The current SIND system also has no real threading or multi-process support. Although when a new process is created (with `fork()`), SIND should be copied along, this hasn't been tested.

For an overview of what functionality is in which files, please consult appendix A. That appendix also contains details on building SIND.

7.2.1 System-Dependent Code

In SIND terms, the system dependent code is that code that is either OS specific, or architecture specific. Basically, that means any code which depends upon low level system behavior, or has inlined assembly code (for state capture, for instance). In the current version of SIND, the bootstrapper and the trap handling code (`SPARCTrap.cc`) are dependent upon both the processor and operating system. The fragment cache is depen-

dent upon the processor (it processes SPARC code), and the signal handling code (signal_handling.c) needs a system with support for POSIX signals (it handles both `signal` and `sigaction`).

7.2.2 System-Independent Code

The rest of the SIND code is intended to be platform independent (insofar as C or C++ can be). The interpreter (`SPARCCPU.cc`) is meant to run on both 64- and 32-bit systems (by emulating a 64-bit datum with a class). `SPARCMMU` is only dependent upon SIND running in the same address space as the target process. Much of the rest of the code is supporting classes for the interpreter (`SPARCInstruction`, etc) or data structure classes (`AddressHash`, etc), and is not specific to any platform. To avoid the annoying unknown bit-width of `int` problem, all the code that needs to specify a given bit width uses the POSIX typing standard (i.e. `uint32_t` for a 32-bit unsigned int).

Appendix A

Technical Details

This appendix describes the nitty-gritty details necessary to take the current SIND code base and build it, or (hopefully) extend or debug it.

A.1 Source Layout and Directory Organization

The base directory contains all the standard GNU files, as well as an out of date configuration script. The `doc` directory contains documents related to SIND, in particular this thesis and the technical reports written about it. All the code is located in the `src` directory. All of the base classes are in `src` the class definitions are in `include` and the implementation files are in `src`. `src` also includes a simple makefile for building these classes. Off of this directory are directories for each architecture. Currently there are only two `PowerPC` and `SPARC`. `PowerPC` contains Dino Dai Zovi's beginnings of a PowerPC interpreter, and `SPARC` contains all of my SPARC and Solaris specific code. The architecture directories are organized similar to the base class directory, class definitions and other header files are in `include`, and the implementations as well as the Makefile are in the `SPARC` directory.

A.2 Where Functionality Resides

In order to get a good handle on the code base it is necessary to create a mapping between the modules described in the design section and the actual files in the directories.

Include Files	
AddressHash.h	definition of the AddrHash data structure
bootstrap.h	function prototypes for bootstrapping
driver.h	definition for driver program
signal_handling.h	prototypes of internal signal handling functions
signal_stuff.h	prototypes of superposed signal functions
SPARCCPU.h	definition of the SPARCCPU (interpreter) class
SPARCDispatch.h	definition of the SPARCDispatch class
SPARCExceptions.h	definitions of error conditions in the interpreter
SPARCFPU.h	definition of the floating point unit
SPARCFragCache.h	definition of the SPARCFragCache class
SPARCInstruction.h	the SPARC instruction class
SPARCInstrFmt.h	a wrapper for a 32-bit integer to readily parse it as an instruction
SPARCMMU.h	definition of the SPARC MMU class
SPARCRegisters.h	definition of registers (both general purpose and special)
SPARCTrace.h	definition of the data structure used to hold traces
SPARCTransformer.h	interface class for transformers
SPARCTrap.h	definition of trap handling functions (for syscall manager)

Table A.1: Include files to modules

Files and Modules

Module	File(s)
Bootstrap	bootstrap.h, bootstrap.c
Interpreter	SPARCCPU.*, SPARCExceptions.h, SPARCFPU.*,SPARCInstrFmt.h, SPARCInstruction.h, SPARCRegisters.h
Trap/Syscall Manger	SPARCTrap.*
Dispatch	SPARCDispatch.*
Transformer	anything implementing SPARCTransformer
Memory Manager	SPARCMMU.*
Fragment Cache	SPARCFragCache.*

Table A.2: modules' source files

A.3 Compilation and Architecture Support

A.3.1 Makefiles

Compilation is currently managed by a set of Makefiles. Each directory that has compilable source will have its own Makefile. The main one for the experimental system is the Makefile in the SPARC directory. This Makefile follows the convention of having the `all` and `clean` targets. `all` will build everything (including files in parent directories) and link it together to create the SIND shared object. After building there should be a file named `libsind.so` in the directory. This is SIND.

A.3.2 Compilation Flags

There are several compilation flags that guide the preprocessing of the SIND source. The first is `SIND_ARCH64`. If this is defined, then it is assumed that the SIND code is running on a 64-bit machine and so can use native 64-bit integers. If this is not defined, then

emulation of 64-bit wide data is done through a class (DoubleWord) using overloaded operators. The second flag is `DEBUG`; if this is defined, scores of debugging statements will be compiled into SIND. This generates voluminous output, and should only be used for debugging. If not defined, SIND's printouts will be limited to a few lines of text when starting up. Also, this means that any extensions made to SIND should respect this convention and enclose debugging statements in `#ifdef DEBUG` conditionals. The next flag of note is `TRACING`, if defined the interpreter will be compiled with tracing support, otherwise all code will be executed in the interpreter. This flag was introduced to allow me to debug interpreter errors even after I had implemented a tracing infrastructure. Another notable flag is `TIMING`; if defined, then SIND will time itself and print out the results. Currently, timing code exists only in the interpreter, and this times the execution in the main interpreter loop (`executeLoop()`). Lastly, there's the `BOGUS` flag, this flag should only be set if compiling a BOGUS version of SIND. The bogus version is used to get a highly accurate snapshot of the machine state for debugging purposes and is discussed in more detail in section 6.2.

A.3.3 Supported Architectures

All the platform independent code should work on any 64-bit platform, and includes enough infrastructure that it could readily be made to work on any 32-bit platform. All the platform dependent code will only work on an UltraSPARC running Solaris2.x and up.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[2] Derek Bruening and Saman Amarasinghe. The DynamoRIO Collaboration. http://www.cag.lcs.mit.edu/dynamorio/.

[3] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, and J. Yates. FX!32 a profile-directed binary translator, 1998.

[4] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zang. Automatic detection and prevention of buffer-overflow attacks. *7th USENIX Security Symposium*, 1998.

[5] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[6] Richard McDougall Jim Mauro. *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, first edition, 5 October 2000.

[7] A. Klaiber. The technology behind Crusoe processors, 2000.

[8] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, first edition, 15 January 2000.

[9] Sun Microsystems. proc - /proc, the process file system. In *SunOS 5.8 Manual*, chapter 4.

[10] Sun Microsystems. ptrace - allows a parent process to control the execution of a child process. In *SunOS 5.8 Manual*, chapter 2.

[11] Sun Microsystems. Solaris linker and libraries guide. Online PDF manual, Part Number 816-0559-10.

[12] Julian Seward and Nick Nethercote. Valgrind, an open-source memory debugger for x86-linux. http://developer.kde.org/ sewardj.