Experiences Constructing a Lightweight SPARC Interpreter for a Dynamic Binary Translator

Trek Palmer
Department of Computer Science
University of New Mexico
Albuquerque, NM
tpalmer@cs.unm.edu

Darko Stefanović
Department of Computer Science
University of New Mexico
Albuquerque, NM
darko@cs.unm.edu

14 March 2003

Abstract

Dynamic binary translation is an important area for compiler research, because additional information available at runtime can substantially improve the effectiveness of optimizations. The difficulty lies in creating a system capable of gathering runtime information without slowing down the running executable. Several such systems have been created (Dynamo, DynamoRIO, FX!32, etc.), but their use presents several problems to the researcher. They are either closed or proprietary, and are often tied to a very specific platform. In this paper we discuss the design of a new, open, cross-platform dynamic binary translation system, SIND. Specifically we discuss the design in general terms, and then focus on the specific implementation of a lightweight interpreter for the SPARC architecture. We explore the many issues involved in building a self-bootstrapping, efficient interpreter.

1 Introduction

In recent years, there has been increased interest in the study of applying the historically static methods of program analysis and optimization to transformations at runtime. With a dynamic profiling system and a lightweight set of optimizations, it should be possible to transform a program at runtime into a more efficient version. Projects such as Dynamo [1] and JIT compilation systems for dynamic languages such as Java have demonstrated the effectiveness of dynamic translation technology. However, research in this field is hampered by a lack of convenient, open tools for experimentation. The Dynamo project itself is a proprietary, in-house system for HP, and its offspring DynamoRIO is closed for legal reasons. Even if DynamoRIO were completely open, the system is so closely tied to the x86 architecture that much of it would have to be rewritten to function on another platform

[Personal Discussions with DynamoRIO maintainer].

SIND is a dynamic translation framework we are implementing to fix some of these problems. SIND aims to be a cross-platform system, by which we mean that the interfaces and architecture will be consistent across platforms, even though some things, of necessity, will be tied to the specific platform. The whole system is designed to be modular and portable with implementations for multiple architectures. When SIND is finished, it will be a useful and capable tool, furthering research into dynamic binary translation. This paper is a discussion of the overall design of SIND, but is primarily focused on the architecture and development of the SIND profiling interpreter for the 64-bit SPARC v9 architecture.

The SIND profiling interpreter itself presented an interesting challenge. It had to be a fully functional interpretation system, but also efficient enough that it didn't slow down overall execution (of the optimized program). Although the SPARC architecture is RISC, which makes writing an interpreter easier, the SPARC has several idiosyncrasies that make interpreter implementation non-trivial (register windows were a particular problem).

2 Architecture of SIND

The SIND dynamic binary translation framework [4] consists of a few main components: A *profiling interpreter* gathers information about the running binary and gathers 'interesting' traces from the binary's execution. A set of *transformers* take these interesting traces and some machine context information and perform analysis and transformations on the traces to produce *code fragments*. Lastly, these code fragments are passed into a *caching system*, which links new fragments together with previously inserted ones and the fragments are then executed directly on the processor. These are the three main parts; additionally, a bootstrapper starts up a target binary and handles its library dependencies, a dispatcher handles intermodule communication, a syscall manager handles system call translation, and a memory access system handles the separation of the SIND system from the target binary.

The SIND system is intended to be modular and was designed in an object-oriented fashion to facilitate modular construction. Each component of the SIND system is a class that implements a well-defined interface (specific to that component). The profiling interpreter, for instance, implements the CPU interface. As the components become more platform-specific, the interface is extended (through inheritance) to include platform information.

SIND is currently implemented as a self-loading library, so that it enjoys inexpensive access to the address space of the target process. The library seizes control of the binary at load time. Interpretation begins at the `_start` symbol of the target binary.

2.1 Comparison of SIND and Dynamo / DynamoRIO

The original work on SIND was inspired by the Dynamo dynamic translation system from HPLabs [1]. It is only natural to want to compare the two systems. Both are dynamic optimization systems and both have an interpreter and a fragment cache. But as one gets deeper into the design, more differences become apparent. For instance, Dynamo was intended to be a vehicle for research, not a production system or a widely-used tool. As a consequence much of the system was highly tied to the platform it was developed on (HPPA running HP-UX). Because so much of the Dynamo system was hard-coded it was extremely non-portable (the developers admitted as much). This is perhaps the largest difference between SIND and Dynamo. In SIND everything is meant to be changeable. This is aided by its object-oriented design and by its open-source status. Because SIND is meant to be a research tool, its components are meant to be replaceable. SIND is also meant to be cross-platform tool, and as such only at the bottom of the inheritance tree are there platform-specific classes.

Because the original Dynamo code was so non-portable, when HP wanted to create a dynamic optimizer for the x86 architecture, they had to rewrite everything. The latest incarnation of this effort is the DynamoRIO system. DynamoRIO itself differs considerably from Dynamo. Firstly, because the x86 instruction set is complicated to decode and emulate, DynamoRIO uses a basic block cache rather than a straight interpreter. Secondly, the entire optimization system was tailored to this basic block cache. Lastly, however, the system was engineered to be much more accessible than the original Dynamo, and as a result has a sophisticated API to allow people to write external programs that use the DynamoRIO engine. The portability of the DynamoRIO system is evidenced by the fact that it runs on both Win32 and Linux operating systems. However, SIND differs from DynamoRIO. DynamoRIO is a monolithic system, and although the API is nicely featured, it only allows external programs to attach to 'hooks' in the system, but not to augment the system with new hooks. Because SIND is open-source and internally modular, replacing portions of SIND is completely possible (in fact, that is how the system is extended). DynamoRIO is also tied to the x86 architecture. SIND, although currently only implemented for the SPARC, is meant to be portable to any architecture.

3 Architecture of the SIND SPARC 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 system is to do runtime binary optimization, it is vital that the interpreter should introduce as low an 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 is for emulating 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 Registers

The interpreter itself 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 `struct`s. 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 sliding 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. We now examine these two requirements.

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.

Maintaining two separate execution stacks requires a bit of hackery. 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 when executing code directly on the processor (either issuing traps or when in the fragment cache) SIND's stack can be `mprotect`-ed to safeguard its contents.

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).

3.2 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.

3.3 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 thinks 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.

[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.

This generates the appropriate operating system action (usually, killing the process).

3.4 Signals and Asynchronous I/O

In the Solaris system there are really only two ways of communication between user and supervisor (kernel) code. One, the system call or trap, has already been discussed. 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 functions, and registers a special handler for all registerable 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.

4 Experiences from the Design and Implementation of SIND

The current SIND system is several generations removed from the first attempt. Initially, we attempted to marry a `procfs`-based bootstrapper with the ISEM full-system emulator [5]. This introduced many problems, because ISEM emulated not just the full processor but an entire system bus (with attached devices). ISEM also was only a 32-bit SPARC v8 interpreter, and so would have to be extended to accommodate 64-bit operands. And lastly, ISEM emulated the running binary's memory system, and so had to run in a separate address space. We were left with two options. Either gut the ISEM system to isolate the user-land portion, or try to capture the exact state of the machine and use it to initialize ISEM. Neither option was particularly attractive, and the ISEM system was almost certainly going to be slower than a custom lightweight interpreter. After getting nowhere with this system for several months we decided to write a new interpreter from scratch.

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 [6] 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 la 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 isn't 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 mprotects 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 isn't 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 hampered diagnosis severely. 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).

The GCC compiler itself introduced problems. We were originally using the gcc 2.95 compiler collection which had somewhat shaky 64-bit support. But 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!

5 Conclusions

Dynamic binary translation is an exciting area of research that has been hindered by a lack of convenient, open tools. SIND is an open-source effort currently targeted at offering a convenient, platform-independent tool-set for research into dynamic binary translation. Central to this effort are lightweight profiling interpreters, which have a peculiar set of design issues. Currently SIND is being implemented for the SPARC v9 architecture, but the overall design of SIND is such that other processors should be easy to support. SIND's implementation consists of a functional lightweight interpreter for the SPARC v9 instruction set, and will soon include trace gathering and fragment caching.

The design issues encountered in the implementation of the SIND interpreter were very different from standard 'application level' design questions. Correctly emulating register windows provided no end of headaches, and tricks such as separate stacks had to be employed to avoid having the user application clobber SIND data. Standard compiler chain tools and debuggers were inadequate for the task.

Acknowledgments. This work was supported in part by NSF grants CCR-0219587 and CCR-0085792, DARPA grant F30602-02-1-0146, Microsoft Research, and Hewlett-Packard. Barney Maccabe wrote and provided us with the ISEM emulator. We especially thank Dino Dai Zovi for help with the original design of SIND.

References

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

[2] D. Bruening, S. Amarasinghe, and E. Duesterwald. Design and implementation of a dynamic optimization framework for Windows. In *FDDO-4 Fourth Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[3] 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(2):56–64, March/April 1998.

[4] D. Dai Zovi, T. Palmer, and D. Stefanovic. SIND: A framework for binary translation. Technical Report TR-CS-2001-38, University of New Mexico, 2001.

[5] A. B. Maccabe. tkISEM, a SPARC version 8 instruction set emulator. http://www.cs.unm.edu/∼maccabe/tkisem/begin.html.

[6] J. Seward. Valgrind, an open-source memory debugger for x86-gnu/linux. http://developer.kde.org/∼sewardj/.