Efficient Static Binary Instrumentation in the Presence of Variable-Length Instructions

Michael Laurenzano, Mustafa Tikir, Laura Carrington, Allan Snavely
San Diego Supercomputer Center
{*michaell, mtikir, laurac, allans*}*@sdsc.edu*

Abstract

Dynamic binary instrumentation toolkits that are in use today produce instrumented code that is at a performance disadvantage to the instrumented code produced by static binary instrumentation toolkits because dynamic binary instrumentation toolkits act at runtime. Therefore in cases where efficiency is paramount it is important to have a binary instrumentation toolkit capable of meeting that need.

In this work we present X86ElfInstrumentor, a static binary instrumentation toolkit for Linux on x86/x86_64 platforms that uses wholesale code relocation in order to remedy the difficulty created by the platforms' use of variable-length instructions. Code relocation of this kind allows the instrumentation tool to reorganize the application code in such a way that it can use the fast but far-reaching constructs to transfer control from the application to the instrumentation code rather than relying on multiple jumps or interrupts for the transfer. Furthermore, the API includes a means of allowing the tool developer to insert hand-coded assembly in a very lightweight way rather than relying solely on the insertion of entire instrumentation functions. These techniques yield very efficient instrumentation tools, with overheads for basic block counting that are an average of 48% of the overhead imposed by Pin, 18% of the overhead imposed by DynamoRIO, 10% of the overhead of Valgrind, and 5% of the overhead of Dyninst.

1 Introduction

Binary instrumention is the act of inserting extra code into a compiled application in order to observe or modify something about its behavior. The data collected from binary instrumentation can be useful for guiding hardware and system design decisions, program debugging, compiler correctness/optimization, program debugging and security testing/verification.

There are two common approaches to binary instrumentation: static binary instrumentation and dynamic binary instrumentation. Static binary instrumentation toolkits have the advantage of usually being able to produce more efficient instrumented code than comparable dynamic binary instrumentation toolkits because static instrumentation introduces only the instrumentation code itself into the run of the application. Static binary instrumentation tools have some disadvantages, however. It is not possible to instrument shared libraries or dyanamically generated code with static binary instrumentation tools. They also provide less flexibility to the tool developer, since any instrumentation code that is inserted into the application will persist throughout the application run. However, there are cases where efficiency is of enough

importance to outweigh these shortcomings in such a way that static binary instrumentation is the desirable paradigm.

x86elfinstrumentor is a static binary instrumentation toolkit for Linux on X86/X86_64 platforms. The goal of x86elfinstrumentor is to provide the ability to build instrumentation tools that produce very efficient instrumented applications. We instrument the binary by placing an unconditional branch at each instrumentation point that transfers control to some instrumentation code. This instrumentation code saves the program state, performs some functions that are determined by the particular instrumentation tool, restores the program state, then returns control to the application. A typical binary instrumentation tool on a platform with fixed-length instructions [1] accomplishes this initial control transfer by replacing a single instruction at the instrumentation point with a branch that transfers control to the instrumentation code. However when instructions are variable-length, as is the case for X86/X86_64, this is not always possible since the branch instruction can be larger than the instruction at the instrumentation point. To address this, x86elfinstrumentor relocates and reorganizes the code for each function to ensure that enough empty space (in the form of *nops*) is available to hold a full-length branch instruction at each instrumentation point.

Instrumented code efficiency is accomplished in several ways. The cost of instrumentation, tasks such as parsing, disassembly and code generation, is taken prior to runtime rather than at runtime as is done by current state of the art binary instrumentation tools for X86/X86_64 [2, 3, **?**]. We relocate the application's functions, which affords us the opportunity to reorganize the code so that we can use large yet efficient instructions to transfer control from the application to the instrumentation code. We also use the concept of an instrumentation snippet, a lightweight hand-written body of assembly code that can be inserted into the application rather than inserting only heavyweight functions to accomplish instrumentation tasks.

2 Implementation

Static binary instrumentation is a process that leaves a modified executable on disk that can be run at a later time. Then when the instrumented executable is run, the extra instrumentation code inserted by the instrumentation tool is run in addition to the normal behavior of the program. In order to insert extra code and data, extra space must be allocated within the executable in a way that it will, at load-time, be treated by the system in a manner appropriate to its purpose. Consider the following. Most compilers emit an executable whose structure is similar to that shown in Figure 2. By convention, most executables use only two loadable segments and certain Linux implementations such as FreeBSD only allow two loadable segments. Thus it is preferrable for us to incorparate instrumentation text and data into the existing text and data segments of the application. The default in most compilers is to place the text segment prior and adjacent to the data segment. We therefore prepend the instrumentation text to the existing text segment[1] and append the instrumentation data to the data segment, which can be seen in Figure 2. This scheme has a further advantage of causing no immediate disturbance to the addresses of the existing text and data segments of the program.

[1]The amount of space allocated prior to the text section is controlled by the linker variable __executable_start. We have seen cases where the system does not provide enough space prior to the text segment by default, in which case we provide a set of tools that produces a modified linker script that provides 128Mb of space.

3 Relocation

The novel use of relocation in our instrumentation strategy stems from the fact that we are performing the instrumentation statically on a platform that uses a variable-length instruction set. A typical strategy used by static instrumentation tools on platforms with fixed-length instruction sets is to replace a single fixed-length instruction at the instrumentation point with a branch instruction that will transfer control to the code produced by the instrumentation tool. This is fairly straightforward to do because by the definition of a fixed-length instruction set, the instruction being replaced and the replacement branch have the same length. Performing static instrumentation in a variable-length instruction set does not afford us this luxury. In X86, an unconditional branch that uses a 32-bit offset requires 5 bytes, whereas some of the instrumentation points that interest us may use only a single byte.

This leaves two options for how to transfer control to the instrumentation code. We must either use a technique entirely distinct from the idea of using a single unconditional branch to execute the control transfer such as multiple shorter jumps or software interrupts, or we must somehow alter the application code so that it can accomodate a single large control instruction that is larger than the original amount of space available at the instrumentation point. A seperate technique for transferring control flow could be to use a series of branches, where the instruction in the instrumentation point is a small branch that transfers control to a larger intermediate branch. We do not consider this method any further because the smallest unconditional branch instruction is 2 bytes in length, making it ultimately a half measure since there are instrumentation points with only a single byte available to them. Another option to consider is the method proposed by the BIRD project **??**. They propose using the single-byte *INT 3* instruction, a single-byte interrupt intended to be used by debuggers to set breakpoints, when a larger branch won't fit within the specified area. This instructions is functionally perfect for static instrumentation because it consumes only a single byte and allows us to transfer control to an arbitrary location by registering an exception handler with the system. We performed a cursory study on this scheme from an efficiency standpoint to determine whether it was worth further investigation. On a small benchmark set, our implementation of using *INT 3* only when 5-byte unconditional branches do not fit at the instrumentation point introduces slowdowns of no less than 100-fold for counting the number of executions of each basic block in the code. As one might expect, this mechanism is unsuitable for efficient instrumentation because the very heavyweight system call conventions are being invoked fairly often.

We use the latter option, reorganizing the code at the function level so that there is enough space at every instrumentation point to accomodate a 5-byte branch. Specifically, the steps we use are as follows:

1. 1. Function Displacement
2. 2. Link Function Entries
3. 3. Branch Conversion
4. 4. Instruction Padding

Figure **??** gives a visual version of this process.

1. Function Displacement: Relocate the contents of the entire function to an area of the text section allocated to the instrumentation tool. Since functions are often packed tightly together, it is generally not possible to expand the size of a function without disturbing the entry points of another function.

2. Link Function Entries: Place an unconditional branch at the former function entry point that transfers control to the new function entry point. Most references to the entry point of a function are in the form of function calls, which routinely are indirect references (ie their value is computed or looked up at runtime) and are difficult to resolve prior to runtime.

3. Branch Conversion: Convert each short conditional branch in the relocated function to the equivalent 5-byte branch instruction. Since the code is being reorganized in the next step which may strain the limits of smaller 8-bit or 16-bit offsets, we convert all branches to use 32-bit offsets so that the targets of each branch will still be reachable without having the need to further reorganize the code. Note that there is some opportunity here to reduce space by using the smallest branch offset size that accomodates the branch, but this is an issue for future work.

4. Instruction Padding: According to the needs of the instrumentation tool, pad the instruction at each instrumentation point with *nop* instructions so that a 5-byte branch can fit.

There are several ways that this process can adversely affect the performance of the application independant of the overhead that will be imposed by inserting any extra instrumentation code. Each function call now has an extra control interruption associated with it since control must be passed first to the original function entry point and then to the relocated function entry point. It is possible that using 32-bit offsets for every branch rather than some smaller number of bits has an overhead associated with it. And since the code is being reorganized and expanded, we might destroy some positive alignment and size optimizations that the compiler might have made on the instructions in the function. We examine the practical overhead seen by these techniques by taking these steps without instrumenting the code for a series of benchmarks. The slowdown is XXXX...

4 Coverage

Code and data can reside together in the text section of a program binary. This is done for a variety of reasons, including the storage of branch target locations (eg for a jump table) or small data structures that provide convenient lookup of certain data such as identifiers, descriptors, or other values.

Correctly determining what parts of the text sections are code and what are data is important. Consider what can happen if we mistakenly treat some data as code. We might choose to modify or relocate the apparent code to serve our instrumentation purposes. Then when the data at this location is referenced, the original program behavior may not be preserved: if we are lucky this will cause application failure due to some unexpected change in control flow or some state condition that is checked by the program. If we are unlucky the corruption might silently manifest itsself by modifying the output of the program. Alternatively consider what can happen if we mistakenly treat some code as data. We then will not try to insert code into this area or we might perform some other type of analysis that should be reserved for data alone. While this is almost certainly preferrable to the situation where we treat data as code, it is ideal to avoid both situations.

To this end, we use the program's symbol table to help us determine which parts of the text sections are functions that are eligible to be subject for our code discovery algorithm. Our code discovery algorithm consists of two phases; control-driven disassembly backed up by linear disassembly. In more detail, the algorithm works as follows:

1. 1. Control-driven disassembly: from a

function's entry point, follow all understandable control paths. If a problem is encountered, fall back to naive disassembly.

2. 2. Naive disassembly: from a function's entry point, disassemble each instruction in the order it appears in the function. If a problem is encountered, give up.

Problems that can be encountered are situations where an unknown opcode is encountered, where control jumps to the middle of an instruction we've already disassembled, or if control leaves the boundaries of the function. In most cases control-driven disassembly is sufficient to disassemble the entirety of a function, and in most cases control-driven disassembly is a straightforward process because control either falls through to the following instruction or the location of a branch target is embedded entirely within the instruction itself. But there are also cases where the an indirect branch is used, where the target resides either at a fixed address (possibly with some offset), the address that resides in a register, or the address that is at a location given by a register. The latter two cases are very difficult to resolve without runtime information because the computation of the target address can be arbitrarily complex and can span function boundaries. Nevertheless, we perform a poophole examination of the previous instructions to the and can determine the address in simple cases.

Fortunately simple calculations are all that most compilers use to determine targets for jump tables, one of the more common uses of an indirect branch. Often an offset is added to a fixed location to determine where the data comprising the branch target resides. Therefore we treat such a fixed address as the first entry in a table whose entries are treated either as addresses or as offsets. We then make an iterative pass over this table to determine the target for each arm of the jump table, stopping when we find a value in the table that yeilds an address that is outside the scope of the function.

PUT EXAMPLE OF GNU COMPILER JUMP TABLE HERE

5 Snippets

6 Results

7 Future

Despite some success in terms of efficiency, there are several more techniques that might make the instrumented code even more efficient. Because we are relocating the text to give ourselves as much space as possible, rather than inserting just a branch that transfers control to the instrumentation code we have the opportunity to inline the instrumentation code itsself in order to reduce or eliminate the control interruptions that otherwise must be taken when inserting instrumentation code.

We could also perform register liveness analysis in order to determine whether there is state that doesn't need to be saved around instrumentation code or to guide the selection of usable registers by the instrumentation tool developer. And similar to Pin, we could perform liveness on the bits of the eflags/rflags register to determine whether it must be saved and restored. Saving and restoring state is a large portion of the overhead associated with performing small tasks in instrumentation snippets.

tool-specific – path counter, xiaofeng's memory instructrion subset

8 Conclusions

References

[1] M.M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. PMaC Binary Instrumentation Library for PowerPC/AIX. In *Workshop on Binary Instrumentation and Applications*. Citeseer, 2006.

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

[3] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 PLDI conference*, volume 42, pages 89–100. ACM New York, NY, USA, 2007.