Efficient Static Binary Instrumentation for X86/X86_64 on Linux

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 toolkits enable insertion of additional code into an executable in order to observe or modify the behavior of application runs. Instrumentation toolkits have been widely used to gather information about the important characteristics of applications to be later used in modeling and optimization of the applications. It has been shown that data gathered from binary instrumentation can be effectively used in guiding hardware and system design, program debugging and correctness, compiler optimizations, performance modeling/prediction, and security verification [CITE PMAC].

There are two main approaches to binary instrumentation: *static* and *dynamic* binary instrumentation. Static binary instrumentation inserts additional code in to an executable and generates a new executable with the instrumentation whereas the dynamic instrumentation inserts additional code at runtime during the execution. The static approach has the advantage of usually being able to produce more efficient instrumented executable compared to the dynamic approach since static instrumentation introduces only the instrumentation code itsself into the executable and includes the instrumentation code in to the text section of the executable. Unlike static instrumentation, dynamic instrumentation inserts additional code in to the program heap and uses the data space as text space. However, static binary instrumentation has disadvantages. It is not possible to instrument shared libraries unless the shared libraries used are instrumented seperately and target executable is informed to use these instrumented libraries. Static instrumentation also provides less flexibility to the tool developer, since any instrumentation code that is inserted persists throughout the application run where as dynamic instrumentation provides means to delete instrumentation code when it is not needed [CITE CODE COVERAGE]. However, there are cases where efficiency is of enough importance to outweigh these shortcomings in static instrumentation [CITE PMAC] in such a way that static binary instrumentation is the desirable paradigm.

In this paper, we instroduce a static binary instrumentation toolkit, *x86elfinstrumentor*, for Linux on x86/x86_64 platforms. The goal of x86elfinstrumentor is to provide the ability to build instrumentation tools that produce efficient instrumented applications. Similar to previous instrumentation libraries[CITE Dyninst], we instrument the executable by

placing an unconditional branch at each instrumentation point that transfers control to instrumentation code. This instrumentation code saves the program state, performs tasks 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 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. Dynamic binary instrumentation is costly because the instrumentation tool must frequently disrupt the application in order to perform tasks such as parsing, disassembly, code generation, and other decisions. This interruption is simply not an issue with static binary instrumentation tools because all decisions and actions are taken prior to runtime. The only cost born at runtime is the direct cost of performing instrumentation-related functions. 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 relying only on heavyweight instrumentation functions to accomplish instrumentation tasks.

The x86elfinstrumentor is open source and available to the public for download along with already implemented several instrumentation tools at `http://blind-review-forbids-real-urls.com` These tools include a function execution counter,a basic block execution counter, and a memory address stream collection tool.

The rest of the paper is organized as follows. Section 1 descries the design and implementation of our static instrumentation toolkit. Section 2 discusses in greater detail several aspects of the toolkit, including the function relocation mechanism and the in-

strumentation snippet concept. Section **??** shows the details of the instrumentation tools included in the package. Section 3.3 presents a comparison and discussion of the performance applications instrumented by out x86elfinstrumentor compared to other state of the art instrumentation toolkits including Dyninst, Pin, Valgrind. Section 4 discusses some ideas for the future of x86elfinstrumentor, and Section 5 concludes.

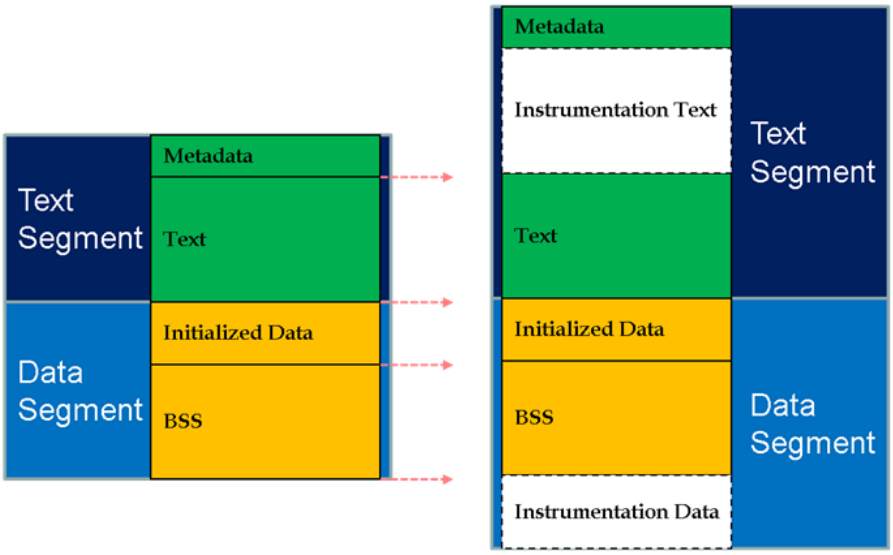
2 Overview

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 1. 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(b). This scheme has the added benefit of causing no immediate disturbance to the addresses of the existing text and data segments of the program.

The instrumentation text contains several code of several kinds. The first contains code that accomplishes the instrumentation task as well as some code to accompany it. When control is transferred from the application to the instrumentation code, it is necessary to maintain the pristine machine state of the application in order to preserve its original behavior. This machine state can contain anything modified by the instrumentation code, but in practice is usually limited to a relatively limited set of registers. The

[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.

Figure 1: (a) and (b) show the prepending of instrumentation text to the existing text, and the prepending of instrumentation data to the existing data respectively.



(a) The two-segment structure of an unmodified ELF file.

(b) The two-segment structure of the ELF file once instrumentation code and data are inserted.

trampoline will save any registers it intends to use, perform the instrumentation task, restore any machine state after the instrumentation task is complete, execute the applications instructions that were displaced by the initial control transfer, finally restoring control to the application. Since we are using an unconditional branch instruction at the instrumentation point, at runtime the instrumentation code has no information about where control was transferred from (as might be the case if we used a more heaviweight call instruction). Hence each instrumentation point uses its own trampoline so that the location of the instrumentation point can be hard-coded into an unconditional branch instruction at the end of the trampoline.

The instrumentation text also includes code to initialize data for use by the instrumentation tool. Recall from Figure 1 that the instrumentation data was appended to the end of the application's data segment, after the application's bss (uninitialized data) section. The initialized data and bss sections of the data segment are usually implemented by declaring the size of the data segment in the executable to be smaller than the size of the data segment in memory. According to the ELF specification, the extra part of any segment whose memory size is greater than its file size should be filled with zeroes by the

loader. Hence most programs just increase the size of the data segment's size in memory by the size of the bss section in order to get a large area that is filled with zeroes and is reserved for uninitialized data. Since we would like to use the area following the bss section for (possibly initialized) data for the instrumentation tool, we can either explicitly include the entire segment's contents in the executable file or we can implicitly reserve this area using the technique described above that is already in use by most programs. Since the bss section can be very large and explicit inclusion of its contents would bloat the instrumented executable file unnecessarily, we use the implicit technique to reserve this section for instrumentatoin data. We therefore temporarily store the instrumentation data with the instrumentation text in the executable, as well as some code to copy it to the appropriate location in the data segment once the program starts.

3 Efficiency

The basic design described in the previous section, while functional, would produce somewhat inefficient instrumented code.

3.1 Function Relocation

The novel use of relocation at the function level 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 2 gives a visual version of this process.

Figure 2: The steps taken in order to prepare a function for instrumentation which collects the memory addresses of an application.

```
0000000008048b10 <foo_func>:
 8048b10:    48 89 7d f8    mov    %rdi,-0x8(%rbp)
 8048b14:    5e             pop %rsi
 8048b15:    75 f8          jne    0x8048b14
 8048b17:    c9             leaveq
 8048b18:    c3             retq
```

(a) The two-segment structure of an unmodified ELF file.

```
0000000004048b10 <_rel_foo_func>:
 4048b10:    48 89 7d f8    mov    %rdi,-0x8(%rbp)
 4048b14:    5e             pop %rsi
 4048b15:    75 f8          jne    0x4048b14
 4048b17:    c9             leaveq
 4048b18:    c3             retq

0000000008048b10 <foo_func>:
 8048b10:    e9 de ad be ef    jmpq   4048b10
 8048b15:    66 66 90          nop
 8048b18:    90                nop
```

(b) The two-segment structure of an unmodified ELF file.

```
0000000004048b10 <_rel_foo_func>:
 4048b10:    48 89 7d f8          mov    %rdi,-0x8(%rbp)
 4048b14:    5e                   pop %rsi
 4048b15:    0f 85 f8 ff ff ff    jne    0x4048b14
 4048b1b:    c9                   leaveq
 4048b1c:    c3                   retq
```

(c) The two-segment structure of an unmodified ELF file.

```
0000000004048b10 <_rel_foo_func>:
 4048b10:    48 89 7d f8          mov    %rdi,-0x8(%rbp)
 4048b14:    90                   nop
 4048b15:    5e                   pop %rsi
 4048b16:    0f 85 f8 ff ff ff    jne    0x4048b15
 4048b1c:    c9                   leaveq
 4048b1d:    c3                   retq
```

(d) The two-segment structure of an unmodified ELF file.

```
0000000004048b10 <_rel_foo_func>:
 4048b10:    e9 fa de db ad       jmpq   0x4049310
 4048b15:    5e                   pop    %rsi
 4048b16:    0f 85 f8 ff ff ff    jne    0x4048b15
 4048b1c:    c9                   leaveq
 4048b1d:    c3                   retq
```

(e) The two-segment structure of an unmodified ELF file.

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...

3.2 Disassembly 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 itsself. 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 peephole 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

3.3 Instrumentation Snippets

4 Results

5 Future Work

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

6 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.