PIX: PMaC's Efficient Binary Instrumentor for Linux on x86 Platforms

Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, Allan Snavely
San Diego Supercomputer Center
9500 Gilman Drive, La Jolla, CA 92037
{*michaell,mtikir,laurac,allans*}*@sdsc.edu*

Abstract

Binary instrumentation enables insertion of additional code into an executable in order to observe or modify the behavior of application runs. There are two main approaches to binary instrumentation: static and dynamic binary instrumentation. In this paper, we present PMaC's instrumentation toolkit (PIX), an efficient static instrumentation toolkit for Linux on x86/x86_64 platforms. PIX is similar to the other toolkits in terms of how additional code is inserted. However, it uses whole function level code relocation in order to remedy the difficulty created by the variable-length instruction set. Code relocation of this kind allows the instrumentation tool to reorganize the application code in such a way that it can use the fast 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 PIX API provides a means to tool developers to insert lightweight hand-coded assembly rather than relying solely on the insertion of entire instrumentation functions. PIX also enables implementation of efficient instrumentation tools, with overheads for basic block counting that are an average of 1.6x less than the overhead imposed by Pin, 4.7x less than the overhead imposed by DynamoRIO, 7.8x less than the overhead imposed by Valgrind, and 75.6x less than the overhead imposed by Dyninst.

1 Introduction

Binary instrumentation toolkits enable insertion of additional code into an executable in order to observe or modify the behavior of application runs. Instrumentation toolkits such as Pin[1], Dynint[2], Valgrind[3], DynamoRIO[4] have been widely used to gather information about the application runs to be later used in modeling and optimization of the applications. It has been shown that data gathered from such binary instrumentation tools can be effectively used in guiding hardware and system design, program debugging and correctness, compiler optimizations, performance modeling/prediction, and security verification [5].

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 execution without any permanent modifications to the executable. The static approach has the advantage of usually being able to produce more efficient executable compared to the dynamic approach since static instrumentation introduces only the instrumentation code itself and includes it into the text section of the executable. Unlike static instrumentation, dynamic instrumentation may insert additional code in to the program heap and use the data section as text space. However, static binary instrumentation has disadvantages. It is not possible to instrument shared libraries unless the shared libraries are instrumented separately and executable is informed to use those libraries. Static instrumentation also provides less flexibility to tool developers, 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 [6]. However, there are cases where efficiency is of enough importance to outweigh these shortcomings in static instrumentation [7] such that static binary instrumentation is the desirable paradigm.

In this paper, we introduce a static binary instrumentation toolkit, *PIX*, for Linux on x86/x86_64 platforms. The goal of PIX is to provide the ability to build instrumentation tools that produce efficient instrumented executable. Similar to previous instrumentation toolkits [2], we instrument the executable by placing a jump instruction at each instrumentation point that transfers control to instrumentation code. This instrumentation code saves the program state, performs tasks that are determined by the instrumentation, restores the program state, then returns control to the application. A typical binary instrumentation tool on a platform with fixed-length instructions [8] accomplishes initial control transfer by replacing a single instruction at the instrumentation point with a jump 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 there may not be enough space for inserting the jump instruction correctly. To address this, PIX 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 in PIX. Dynamic binary instrumentation incurs additional runtime cost because the instrumentation tool performs additional tasks such as parsing, disassembly, code generation, and other decisions at runtime. This is simply not an issue with static binary instrumentation tools as all decisions and actions are taken prior to runtime. The only cost born at runtime is the direct cost of performing instrumentation-related functions. In PIX, we relocate the functions in the executable, 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, as in Dyninst, 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 PIX toolkit is open source and available to the public for download at `http://blind-review-forbids-`
The distribution includes already implemented tools including a function execution counter, a basic block execution counter, and a memory address stream collection tool.

The remainder of the paper is organized as follows. Section 1 describes the design and implementation of PIX. Section 2 discusses several aspects of the toolkit, including the function relocation mechanism and the instrumentation snippet. Section **??** shows the details of the instrumentation tools included in the package. Section 3.3 presents a comparison of the performance of applications instrumented by out PIX to the other state of the art instrumentation toolkits including Dyninst, Pin, Valgrind and DynamoRIO. Section 4 discusses the future of PIX, and Section 5 concludes.

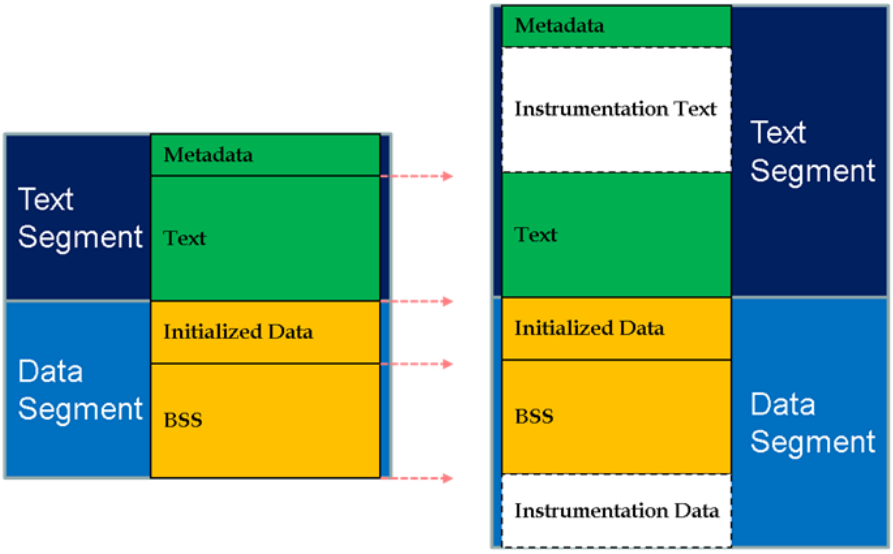
2 Overview

Static binary instrumentation generates a modified executable that can be run at a later time. When the instrumented executable is run, the extra instrumentation code is run in addition to the original code of the program. In order to insert additional code and data, additional 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.

Most compilers produce an ELF executable whose structure is similar to that shown in Figure 2(a). By convention, most executables use only two loadable segments and some of the Linux implementations, such as FreeBSD, only allow two loadable segments. Thus, it is preferable for us to incorporate 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 (as shown 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.

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 structure of an unmodified typical ELF file.

(b) The structure of instrumented ELF file.

The instrumentation text contains several types of additional code. 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 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 but in some cases includes some information about the call stack. The code snippet, called a *trampoline* [2], saves any machine state that will be destoryed, performs the instrumentation task, restores the machine state after the instrumentation, executes the original instructions that were displaced by the initial control transfer, finally restoring control to the application. Since we are using a jump instruction at the instrumentation point, the instrumentation code has no information about where control was transferred from (as might be the case if we used a more heavyweight 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.

Since some instrumentation tasks may need to include additional data for instrumentation code inserted, PIX provides mechanisms to add and initialize additional data in to the executable. The instrumentation text also includes code to initialize this additional 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 uninitialized data section (BSS section) in order to preserve the application's original addresses. The initialized data and BSS sections of the data segment are usually implemented by declaring the size of the data segment

[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 up to 128MB of space.

in the executable to be smaller than the size of the data segment in memory. According to the ELF specification[9], 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 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 additional 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 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 executable file, we use the implicit technique to reserve this section for instrumentation 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

3.1 Function Relocation

The 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 and it may not be always possible to instrument an arbitrary point in the executable due to the lack of enough space for jump instruction to the instrumented code. 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 instrumentation code. This is fairly straightforward because by the definition of a fixed-length instruction set, the instruction being replaced and the replacing jump have the same length. Performing static instrumentation in a variable-length instruction set does not afford us this luxury. In x86 platforms, a jump instruction that uses a 32-bit offset requires 5 bytes, whereas for some of the instrumentation points that interest us, there may not be enough space due to several reasons including the size of basic block or whether it falls in between the target of another jump/branch instruction in the original text of the executable.

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 and traps, or we must somehow alter the application code so that it can accommodate a single large control instruction that is larger than the original amount of space available at the instrumentation point. A separate 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. Besides, this technique would require additional space to be inserted in the text section in the close proximity of the instrumentation points. Another option to consider is the method proposed by the BIRD project **??**, which proposes the use of the single-byte *INT3* 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 instruction 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 *INT3* only when 5-byte unconditional branches do not fit at the instrumentation point introduces slowdown of no less than 100-fold

for even a simple task of counting the number of executions of each basic block in the code. As one might expect, this mechanism is unsuitable for efficient instrumentation since the very heavyweight system call conventions are being invoked fairly often.

In PIX, we use the reorganization of the code at the function level so that there is enough space at every instrumentation point to accommodate a 5-byte branch. Specifically, the steps in whole function relocation includes function displacement, linking function entry points, branch conversion and instruction padding. **(COMMENT: What about the targets of branches in to the middle?)** Figure 2 presents the flow of information on this process with a simple example function in the original text section of an executable.

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.

Function Displacement relocates the contents of the entire function to an area of the text section allocated for 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 using the original location of a function. *Linking Function Entries* places an unconditional branch at the former function entry point that transfers control to the new relocated function entry point. Most references to the entry point of a function are in the form of function calls, which routinely are indirect references (i.e. their value is computed or looked up at runtime) and are difficult to resolve prior to runtime. *Branch Conversion* converts 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 opportunity here to reduce space by using the smallest branch offset size that accommodates the branch, but we chose to use a single mechanism to simplify the implementation and optimize the space usage in the future. *Instruction Padding* pads the instruction at each instrumentation point with *nop* instructions so that a 5-byte branch can fit according to the needs of the instrumentation tool.

There are several ways that whole function relocation may adversely affect the performance

of the instrumented executable independent of the overhead that will be imposed by the additional 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. To quantify the impact of whole function relocation on the performance of applications, we conducted several experiments where we generated executables in which only functions are relocated and no instrumentation code is inserted and compared their performance to the original executables. Thus, we examine the practical overhead seen by these techniques by taking these steps without instrumenting the code for a series of benchmarks. The results of these experiments are described in detail in Section SSSS but in the average the slowdown incurred by whole function relocation is XXXX in the average, which is negligible.

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 look-up 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 itself 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 preferable 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 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 yields an address that is outside the scope of the function.

PUT EXAMPLE OF GNU COMPILER JUMP TABLE HERE

3.3 Instrumentation Snippets

In most instrumentation tools, the tasks accomplished by the instrumentation tool are accomplished by allowing the user to transfer control from instrumentation points in the application to instrumentation functions provided by the user, typically via a shared library or some other type of object code. Since these instrumentation functions are delivered via a shared library or other object code, the instrumentation tool developer has the advantage of the use of a software devlepment toolchain and can write the code in a language that compiles to object code. This delivery mechanism is, however, somewhat heavyweight. Each time the instrumentation tool needs to accompish any task, the overhead of a function invokation must be born. In cases where efficiency is important and where the instrumentation task is relatively small, it can be desirable to insert small sequences of assembly code to perform a task rather than relying entirely on more heavyweight instrumentation functions.

Consider the example of an instrumentation point where we wish to update a counter that resides in memory, which could be of particular use for basic block or instruction counting tools. In order to accompish this task with an instrumentation snippet, we transfer control to the instrumentation point's trampoline which will save athe flags register, update the counter in memory (this does not even require a register), restore the flags register, then transfer control back to the application. Using an instrumentation function, prior to performing the task the trampoline must save the flags register, any registers used by the function, and perform stack protection in some cases. The larger costs associated with using an instrumentation occur because the instrumentation function requires at least 2 more control flow transfers to enter and exit the function. Furthermore these control flow transfers generally use the call/return paradigm, which in addition to changing the application's program counter will also store and retreive information about the function call site onto the stack. There will also be some overhead associated with setting up a stack frame for the instrumentation function. The use of the instrumentation function is also likely to pollute the instruction cache more than using an instrumentation snippet. For an instrumentation snippet the application code must contend with the trampoline code only, whereas using an instrumentation function puts the function code into contention with the other two as well.

These factors show us that instrumentation functions tend to be a much more heaviweight solution, which is not appropriate for certain kinds of instrumentation tasks. This technique can allow us to more efficiently gather asynchronous program information, which intuitively can be thought of as any information that could be dumped to disk an processed offline. The authors of [10] show that using lightweight instrumentation snippets to buffer information which is later processed by more heavyweight instrumentation functions in batches can be an efficient yet entirely lossless way of processing asynchronous program information. The avaialability of instrumentation snippets gives tool authors the flexibility to choose between these and instrumentation functions depending on their efficiency and software engineering needs.

The stack requires protection from the instrumentation function because compilers will often

optimize a leaf function by not explicitly creating a stack frame for the local function data to operate in. This optimization is safe for the application because during its normal execution a leaf function will never call another function and thus its errant stack contents can never be smashed. In the case of an instrumentation tool that calls an instrumentation function from a leaf, this guarantee no longer holds. Thus we must protect the area above the stack when we call an instrumentation function from a leaf function. During the disassembly of a function we note whether it is a leaf (ie whether it contains any call instructions), and then during instrumentation we automatically protect the stack contents for any instrumentation function calls that are made by incrementing the stack pointer by a large fixed amount, which has the effect of giving the leaf function a large stack frame while the instrumentation function's stack frame uses the stack.

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] 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.

[2] B. Buck and J.K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317, 2000.

[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.

[4] D.L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[5] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *IEEE Workshop on Workload Characterization*, volume 2001, 2001.

[6] M.M. Tikir and J.K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96. ACM New York, NY, USA, 2002.

[7] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[8] 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.

[9] T.I. Standard. Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee, May*, 1995.

[10] X. Gao, B. Simon, and A. Snavely. ALITER: An asynchronous lightweight instrumentation tool for event recording. *ACM SIGARCH Computer Architecture News*, 33(5):33–38, 2005.