PEBIL: Efficient Static Binary Instrumentation for Linux

Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, Allan Snavely
Performance Modeling and Characterization Laboratory
San Diego Supercomputer Center
{michaell,mtikir,lcarring,allans}@sdsc.edu

***Abstract*—Binary instrumentation facilitates the insertion of additional code into an executable in order to observe or modify the executable's behavior. There are two main approaches to binary instrumentation: static and dynamic binary instrumentation. In this paper we present a static binary instrumentation toolkit for Linux on the x86/x86_64 platforms, PEBIL (PMaC's Efficient Binary Instrumentation Toolkit for Linux). PEBIL is similar to other toolkits in terms of how additional code is inserted into the executable. However, it is designed with the primary goal of producing efficient-running instrumented code. To this end, PEBIL uses function level code relocation in order to insert large but fast control structures. Furthermore, the PEBIL API provides tool developers with the means to insert lightweight hand-coded assembly rather than relying solely on the insertion of instrumentation functions. These features enable the implementation of efficient instrumentation tools with PEBIL. The overhead introduced for basic block counting by PEBIL is an average of 65% of the overhead of Dyninst, 41% of the overhead of Pin, 15% of the overhead of DynamoRIO, and 8% of the overhead of Valgrind.**

I. Introduction

Binary instrumentation toolkits insert additional code into an executable in order to observe or modify the behavior of application runs. Instrumentation toolkits such as Pin[1], Dyninst[2], Valgrind[3] and DynamoRIO[4] have been widely used to gather information about application runs. It has been shown that data gathered from instrumentation tools can be effectively used in guiding hardware and system design[5], program debugging and correctness[6], program optimization[7], security verification[8], and performance modeling/prediction[9].

There are two main approaches to binary instrumentation: *static* and *dynamic*. Static binary instrumentation inserts additional code and data into an executable and generates a persistent modified executable whereas the dynamic instrumentation inserts additional code and data during execution without making any permanent modifications to the executable. The static approach can be advantageous because it usually results in more efficient executables when compared to the dynamic approach. This is a result of the fact that static binary instrumentation introduces only the instrumentation code itself. With dynamic binary instrumentation, additional overhead is introduced because the instrumentation tool must perform additional tasks such as parsing, disassembly, code generation, and making other decisions at runtime. This is simply not an issue with static binary instrumentation tools because all decisions and actions are taken prior to runtime. The only cost taken at runtime is the direct cost of performing additional instrumentation.

However, static binary instrumentation has some disadvantages. It is not possible to instrument shared libraries unless the shared libraries are instrumented separately and the executable is modified to use those instrumented libraries. Static instrumentation also provides less flexibility to tool developers since any instrumentation code persists throughout the application run while dynamic instrumentation provides the means to delete or modify instrumentation code as needed [10]. However, there are cases where the importance of efficiency is enough to outweigh other considerations [11] so that static binary instrumentation is the desirable paradigm. For example large parallel applications in the Data Center or High Performance Computing domain may run for hours and use hundreds or even thousands of processors. In such cases, instrumented code that runs with lower overhead may make it practical to collect information about a run while still meeting deadlines. Furthermore, within these realms larger overheads may lead to insufficient resources to both run a production job and to thoroughly measure the properties of such a job.

In this work we present PEBIL, **P**MaC's **E**fficient **B**inary **I**nstrumentation Toolkit for **L**inux. The goal of PEBIL is to provide a toolkit that enables the construction of instrumentation tools that produce an efficient instrumented executable. Similar to previous instrumentation toolkits [2], PEBIL instruments the executable by placing a branch instruction at each instrumentation point in the application that transfers control to the instrumentation code. This instrumentation code saves program state, performs tasks requested by the instrumentation tool, restores program state, and then returns control to the application. A typical binary instrumentation tool on a platform with fixed-length instructions [12] accomplishes this initial control transfer by replacing a single instruction at the instrumentation point with a branch that transfers control to the instrumentation code.

When instructions are variable-length, however, this strategy is not always possible since there may not be enough space to correctly insert such a branch instruction. To address this, PEBIL relocates and transforms the code for each function to ensure that enough space is available to hold a full-length branch instruction at any instrumentation point. This method of function relocation enables transformation of the code so that PEBIL can use longer-range yet efficient branch instructions to transfer control from the application to the instrumentation code. PEBIL also allows for the insertion of instrumentation *snippets*, which are lightweight hand-written bodies of assembly code that can be used to perform instrumentation tasks, rather than relying solely on

heavyweight instrumentation functions.

The PEBIL toolkit, along with other accompanying tools and documentation, is open source and available to the public for download at [13]. The PEBIL distribution contains several instrumentation tools including a function execution counter, a basic block execution counter, and a cache simulation tool that consumes the memory address stream of an application run. Each of these tools is built on top of an API that provides both enough low-level detail to allow the tool developer to get involved in the details of instrumentation and enough high-level capability to allow the tool builder to ignore these details if he wishes. These three instrumentation tools, which are provided with the distribution, are implemented in less than 700 lines of C++ code.

The remainder of the paper is organized as follows. Section II describes the basic design and implementation of PEBIL. Section III discusses several aspects of the toolkit that are related to efficiency. Section IV presents some experiments that expose the performance penalties imposed by instrumenting applications with PEBIL, as well as a comparison of PEBIL to other binary instrumentation toolkits for x86. Section V discusses the future of PEBIL, Section VI discusses other popular instrumentation toolkits that are related to PEBIL, and Section VII concludes.

II. Design and Implementation

PEBIL is designed to instrument ELF executables that run on the Linux/x86 platforms, including both x86 and x86_64. There are several challenges that must be addressed by any instrumentation tool in this setting, the largest of which include how to correctly interpret the information found in the text segment of the application and how to organize the extra information needed by an instrumentation tool.

A. Application Code and Data Discovery

When compilers explicitly produce a text and data segment for an ELF application, most take the default action of placing the text segment prior and adjacent to the data segment. However in any ELF executable data can also be intertwined with code in the text segment of the executable, which in practice is done for several reasons. These reasons include the storage of branch target locations (e.g. for a jump table that results from a switch statement) or the storage of small data structures that provide convenient and efficient lookup of data such as identifiers and descriptors. For the sake of correctness of the instrumented executable, it is necessary to identify the parts of the text section that constitute code and the parts that constitute data. Mishandling of the data in the text section as code may result in instrumented application behavior that differs from the original behavior, especially if the instrumentation tool modifies or relocates some part of that data to serve the needs of the instrumentation tool. Such a change in program behavior may cause outright application failure due to some unintended change in the control flow or state of the program. Alternatively, if code is mistakenly treated as data in the text section, instrumentation might not be inserted or analysis performed that should be reserved for data alone.

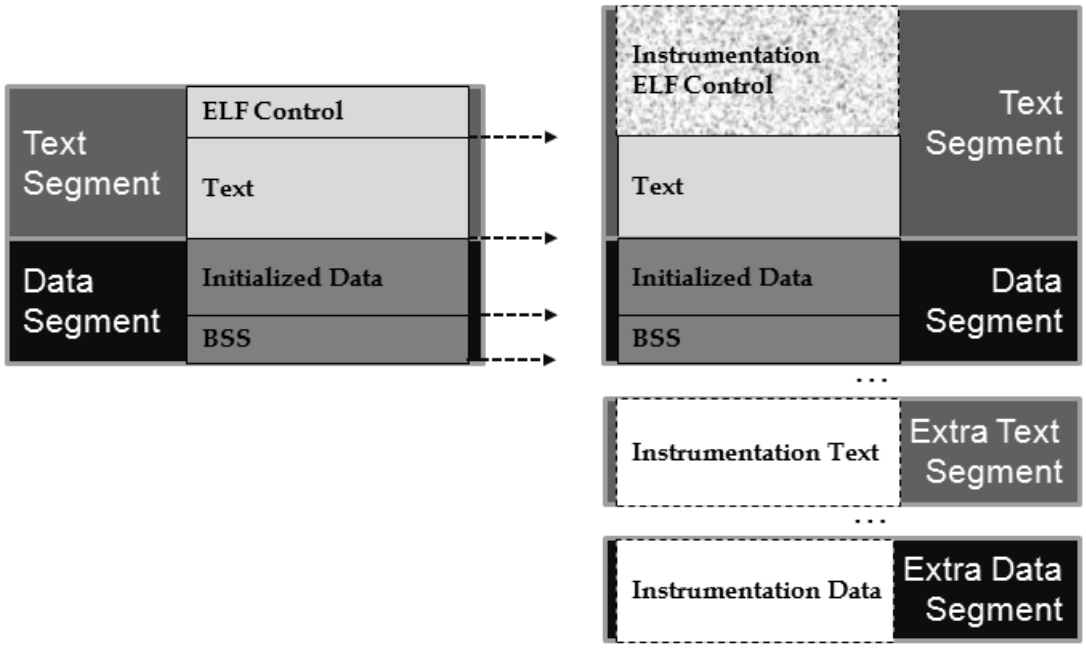
PEBIL currently uses a code discovery algorithm that operates on a per-function basis. In order to determine which parts of the text sections are functions that contain code, PEBIL uses the program's symbol table entries[1] to guide it to each function's entry point. Note that it is possible and in some cases preferable to use techniques that do not rely on the information found in the symbol table in order to perform code discovery. However, when employed statically, these techniques tend to result in less complete disassembly coverage because many control flow instructions use runtime information to determine their targets (for example, as the result of using a function pointer).

The code discovery algorithm being used consists of two possible phases: (1) *control-driven disassembly* backed up by (2) *linear disassembly*. During the control-driven disassembly phase PEBIL follows the control flow through a function, beginning at its entry point. If a problem is encountered during disassembly, PEBIL assumes that some part of the disassembly is incorrect and falls back to the second, more naive, linear disassembly phase. During the linear disassembly phase, each instruction is disassembled in the order it appears in the function, again beginning at the function's entry point. If a problem is found, the function is tagged as uninstrumentable and the disassembly of the function is left incomplete and disregarded for further instrumentation.

Problems that can be encountered during both phases of code discovery are situations where an undefined opcode is encountered, where control appears to fall into the middle of an instruction PEBIL has already disassembled, or if control leaves the boundaries of the function via a traditional branch (not a call) instruction. In most cases control-driven disassembly is sufficient to disassemble the entire function, and at most instructions determining the control target is straightforward because control either falls through to the following instruction or the location of another possible control target is encoded entirely within the instruction itself. In more challenging cases an indirect address is used by a control instruction, where the target resides either at a fixed address in memory (possibly with some offset), in a register, or at a location given by a register. The cases where target computation uses a register value can be difficult to resolve without runtime information since the computation of the target address can be arbitrarily complex and can span control boundaries. Nevertheless, PEBIL performs a peephole examination of the preceding instructions and is able to determine the target address of the indirect branch in most cases. This 2-phase disassembly algorithm correctly disassembled an average of 99.0% of the bytes in the SPEC CPU2000 Integer benchmarks.

One of the more common uses of the indirect jump is as part of a jump table. Fortunately most compilers use relatively simple calculations to determine targets for jump tables. Usually an offset is added to a fixed memory address to determine where the data comprising the branch target resides. Therefore if a peephole examination of the preceding instructions reveals a fixed memory address nearby, this address is treated as the first entry in a table whose remaining entries are considered to be either addresses or offsets that comprise the entries of a jump table. PEBIL makes an

[1]The use of the program's symbol table requires that the program be compiled with debugging information, `-g` for most compilers.



(a) Layout of an unmodified ELF file.

(b) Layout of a PEBIL-instrumented ELF file.

Fig. 1. (a) and (b) show that the text required by an instrumentation tool is prepended to the application text and the data required by an instrumentation tool is appended to the data application data.

iterative pass over this table to determine the target address for each entry in the jump table, stopping when it finds a value in the table that yields a target address that is outside the function. Once found and treated correctly, the jump table code and targets can be integrated into the control flow graph of the function, where it and the data that accompanies it can be modified to accommodate the insertion of instrumentation code.

B. Instrumentation Code and Data

Instrumentation generally requires the use of code and data that are not part of the original executable. In order to insert additional code and data into an executable, additional space needs to be allocated within the executable in such a way that they will, at load time, be treated as code and data respectively. Most compilers produce an ELF executable whose structure is similar to that shown in Figure 1(a). To accommodate the code and data needed for instrumentation, PEBIL appropriates a segment for instrumentation text and a segment for instrumentation data. It also extends the existing text segment in a way that allows some of the existing control structures of the ELF file to be upgraded and expanded to contain the extra information needed for instrumentation. This scheme, demonstrated in Figure 1(b), has the added benefit of causing no immediate disturbance to the original application's code and data, greatly simplifying the implementation of any code modification undertaken later on during the instrumentation process.

The code introduced by PEBIL to perform instrumentation serves several functions. It saves any machine state that can be destroyed by the instrumentation, performs the instrumentation task, restores the machine state after the instrumentation task is completed, and finally restores control to the original code. When control is transferred from the application to the instrumentation code, it is necessary to maintain the machine state of the application since the instrumentation code may use and destroy some of that state and the original application behavior must be observed. This machine state can contain anything modified by the instrumentation code, but in practice is usually limited to a relatively limited set of registers and some information about the call stack.

Since instrumentation tools may also need additional data to support the needs of the instrumentation tool, PEBIL provides mechanisms to insert and initialize additional data within the executable. These additional code and data that are used by the instrumentation tool are included in the extra text and data segments of the ELF file respectively, as shown in Figure 1(b).

III. Efficiency of Instrumented Code

The goal of PEBIL is to provide a toolkit that enables the construction of instrumentation tools that produce efficient instrumented executables. Several techniques are employed to accomplish this. Fast constructs are used to get control to and from the instrumentation code, which requires the application code to be relocated and transformed in order to accommodate them. PEBIL also supports the use of lightweight instrumentation snippets that can be used in place of instrumentation functions, as well as the inlining of these snippets in order to avoid potential control interruptions around their execution at instrumentation points.

A. Code Relocation and Transformation

On platforms with fixed-length instruction sets, a common strategy used by static instrumentation toolkits to transfer control from application code to instrumentation code is to replace a single instruction at the instrumentation point with an unconditional branch instruction that performs the transfer.

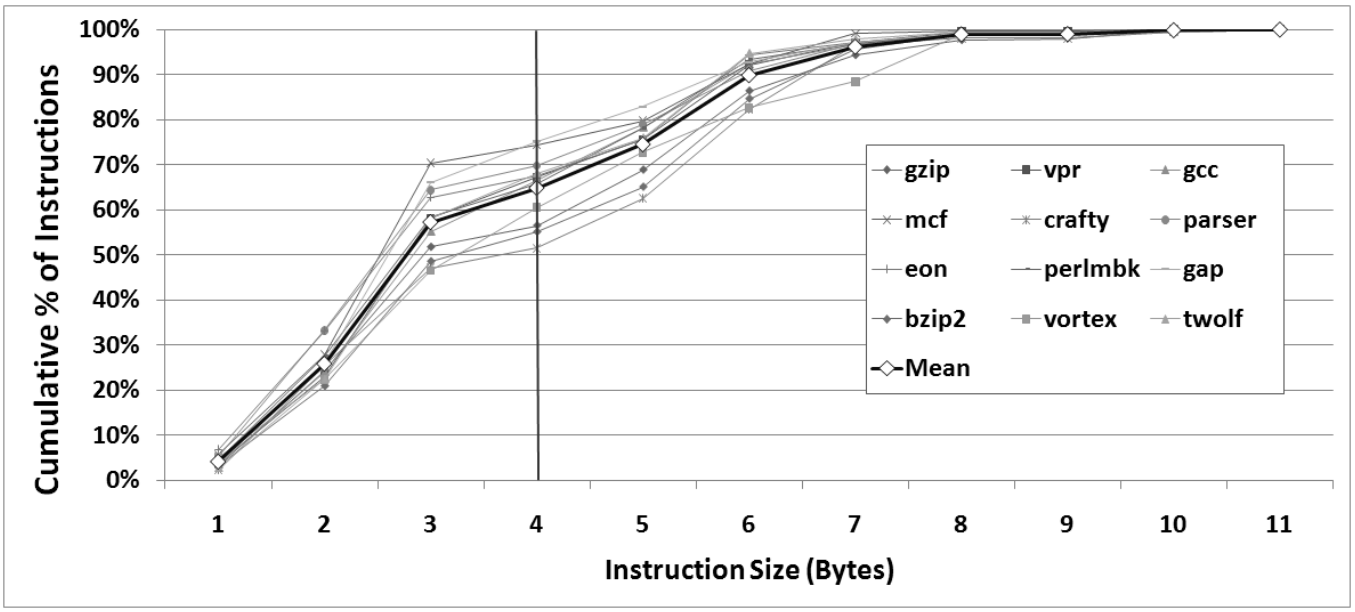


Fig. 2. Instruction sizes for the SPEC CPU2000 Integer benchmarks presented on a cumulative basis.

This is fairly straightforward because, by the definition of a fixed-length instruction set, the instruction being replaced and the replacing branch instruction have the same length. The use of relocation at the function level in PEBIL stems from the fact that instrumentation is being performed statically on a platform that uses a *variable-length* instruction set. The presence of variable-length length instructions means that it may not always be possible to instrument an arbitrary point in the application using this traditional replacement technique since the amount of space available at the instrumentation point is often less than the amount of space needed by a branch instruction that is large enough to reach the instrumentation code.

In x86 platforms, an unconditional branch instruction that uses a 32-bit offset requires 5 bytes. However, for many instrumentation points of interest there may not be enough space to hold a 5-byte branch instruction because the instrumentation point itself (which might be an instruction or a basic block) is smaller than 5 bytes. Figure 2 shows a breakdown of the sizes of instructions for the SPEC CPU2000 Integer benchmark suite. This figure shows that between 52% and 75% of the instructions in these applications are smaller than 5 bytes. In fact an average of 65% of instructions are smaller, which indicates that simply replacing an instruction with a branch to instrumentation code is not sufficient to allow the instrumentation of arbitrary locations in the application code.

This leaves two techniques that can be used to transfer control to the instrumentation code. The first alternative is to use a series of branches, perhaps where the instruction at the instrumentation point is a small branch that transfers control to a larger intermediate branch which in turn delivers control to the instrumentation code. This method is unsatisfactory because the smallest traditional branch instruction available on the x86 platform is 2 bytes in length, yet there are instrumentation points with only a single byte available to them. Refer again to Figure 2, which shows that an average of 4% of instructions use a single byte. Furthermore, this technique requires additional space to be available in close proximity to the instrumentation points since these smaller 2-byte branches are also short reaching, and such space is unlikely to be available since functions are often packed tightly together within the application text.

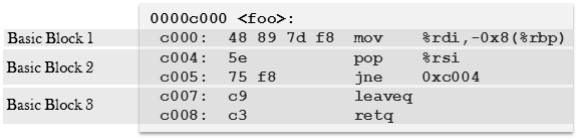
Another option is the method proposed by the BIRD project [14], which is to use an interrupt instruction (`int3`) when a larger traditional branch does not fit at the instrumentation point. This instruction is functionally perfect for instrumentation because it requires only a single byte and allows us to transfer control to an arbitrary location by using the exception handling facilities provided by the system. However, it is unsuitable for doing so efficiently because the heavyweight system call conventions must be invoked, however infrequently.

PEBIL uses relocation and reorganization of the code at the function level in such a way that that enough space is available to accommodate a 5-byte branch at each instrumentation point. Specifically, the steps used by PEBIL to relocate the application's functions and prepare them for instrumentation are as follows. Figure 3 shows how this process looks when performed on a trivial function in order to prepare the function for instrumentation at every basic block.

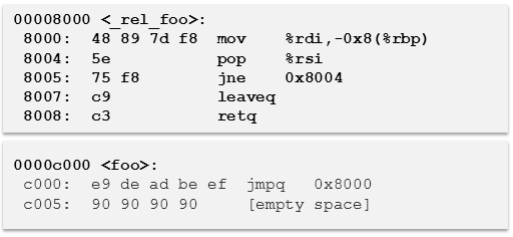
1) *Function Displacement + Entry Point Linking*
2) *Branch Conversion*
3) *Instruction Padding*
4) *Instrumentation*

Figure 3(a) shows the contents of the function prior to any relocation/instrumentation activities.

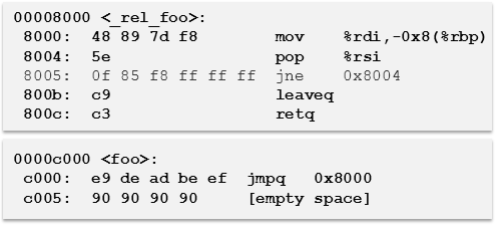
Function Displacement + Entry Point Linking, shown in Figure 3(b), relocates the contents of the entire function to an area of the text section allocated for use by the instrumentation text. This is done because functions are often packed tightly together. As a result it is generally not possible to leave a function's entry point undisturbed and to expand its size in a straightforward way without disturbing the entry



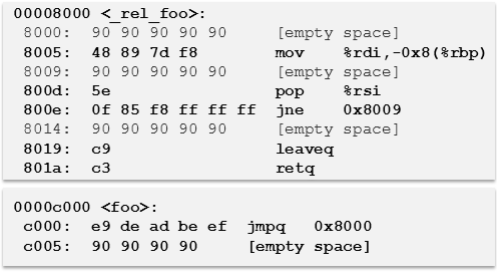
(a) An unmodified application function.



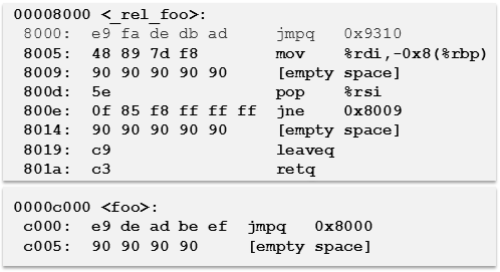
(b) The application function after it has been relocated and the old function entry has been linked to it.



(c) The application function after the branches have been converted to use 32-bit offsets.



(d) The application function after it has been padded with 5 bytes at each instrumentation point.



(e) The application function after a single basic block (Basic Block 1) has been instrumented.

Fig. 3. The steps taken in order to prepare a function for instrumentation that will be inserted at every basic block.

point of another nearby function. The original entry point of the function is then linked to the new location by inserting an unconditional branch at the original function entry to transfer control to the displaced function entry. Linking is done in this fashion because 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 can be difficult to resolve without runtime information.

Branch Conversion is shown in Figure 3(c). The code is reorganized in the following step, which may strain the limits of smaller 8-bit or 16-bit offsets. Therefore all branches are converted to use 32-bit offsets so that the targets of each branch will still be reachable without the need to make further changes to the code. Note that there may be some opportunity here to reduce space by using the smallest branch offset size that accommodates the branch, but currently a single unified technique is used to simplify the implementation. The experimental evidence shown in Section IV indicates that the opportunities for improving the efficiency for this step are minimal.

Instruction Padding, seen in Figure 3(d), pads each instrumentation point with enough empty space so that a 5-byte branch can be accommodated.

Instrumentation replaces the instructions at each instrumentation point with a branch that transfers control to the instrumentation code, which is shown in Figure 3(e).

There are several ways that this proposed method for preparing the application code for instrumentation can adversely affect the performance of the instrumented executable aside from of the overhead that will be imposed by the 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 subsequently to the relocated function entry point. In addition it is possible that using 32-bit offsets for every branch rather than some smaller number of bits has an overhead associated with it. Finally since the code is being reorganized and expanded, some positive alignment and size optimizations that the compiler might have made on the instructions in the function might be destroyed.

To quantify the impact of this relocation method on the performance of an executable, for the SPEC CPU2000 Integer Benchmarks, we generated executables in which the executable is set up for instrumentation (extra text and data segments are present but unused and any extra ELF control information is added), functions are relocated, and branches are converted to use 32-bit offsets. Note that this experiment does not measure the destruction of alignment that will occur when 5 bytes are reserved for a control

instruction at each instrumentation point. It would indeed be very difficult to include this effect in our experiment without including either some extra control overhead or extra overhead that would come from decoding/executing the innocuous instructions that fill in for those bytes. The average overhead for the instrumentation-prepared SPEC CPU2000 Integer benchmarks on the reference input is just 1.2%, with a maximum overhead of 4.8%. Thus the overhead incurred by function relocation and code transformation in PEBIL in order to accommodate single 5-byte branch instructions is well within reason and does not represent a significant hurdle for efficiency of the instrumented code.

B. Instrumentation Snippets

In many instrumentation toolkits, the tasks performed by the instrumentation tool are accomplished by allowing the user to transfer control from an instrumentation point in the application to an instrumentation function provided by the user, typically via a shared library or some object code. Since these instrumentation functions are delivered via a shared library or other object code, the instrumentation tool developer has the advantage of being able to use a software development toolchain to write the instrumentation code in a language that compiles to object code. However, from an efficiency standpoint the instrumentation function is heavyweight due to the overhead of performing a function invocation including saving the complete machine state for all possibles cases that occur in the function. In cases where efficiency is important, it can be more desirable to insert small sequences of assembly code to perform a task and only save the small subset of machine state that will be affected rather than relying entirely on instrumentation functions and the more costly state preservation required for their correct use.

Most of the state preservation needed around instrumentation code is in the form of register saving and restoring, but in the case of the x86_64 architecture some of it comes in the form of protecting the application call stack from the instrumentation function. The call stack requires protection from the instrumentation code during execution because the x86_64 ABI allows for a *red zone*, which is an area of 128 bytes above the top of the stack that cannot be modified by signal or interrupt handlers and thus can be used by the application. In practice, this feature is used by compilers within leaf functions that only use a small amount of stack space in order to save time by not explicitly creating a stack frame. Thus, the area above the stack needs to be protected when an instrumentation function is called from a leaf function since the function call may overwrite some data on what appears to be the top of the stack but could be in use by the leaf function. PEBIL protects the stack contents from an instrumentation function call by incrementing the stack pointer by at least 128 bytes prior to modifying the stack in instrumentation, which has the effect of giving the leaf function the appearance of a 128 byte stack frame while the instrumentation code is running.

A good illustration of the use of an instrumentation snippet is at an instrumentation point where the desired outcome for the instrumentation tool is to increment a counter that resides in memory. In order to accomplish this task with an instrumentation snippet, control is transferred to the instrumentation code which will save the `flags` register, update the counter in memory, restore the `flags` register, then transfer control back to the application. If instead one used an instrumentation function, prior to performing the task the tool must save the `flags` register, any registers used by the function, and possibly perform stack protection. It also requires at least two more control transfers in order to enter and exit the instrumentation function. Furthermore these control flow transfers generally use the call/return paradigm, which in addition to changing the application's program counter also store and retrieve information about the function call site onto the stack and thus have some extra overhead associated with them.

Furthermore, the use of an instrumentation function is also likely to pollute the instruction cache more than using a compact instrumentation snippet. For an instrumentation snippet the application code must contend with the snippet code only, whereas using an instrumentation function puts the application code in contention with the application and the intermediate code needed to save/restore machine state and transfer control to the function. Since instrumentation functions tend to be more heavyweight than instrumentation snippets, using snippets rather than functions whenever possible allows PEBIL to gather program information quickly, particularly when the task being performed in instrumentation is small.

C. Instrumentation Inlining

The fact that PEBIL relocates functions yields a practically unlimited amount of space that can be used in order to further optimize the execution of instrumentation-related tasks by inlining them into the function code rather than inserting an instruction to transfer control to code that performs the same task. Inlining eliminates the overhead that goes along with the execution of control transfers associated with running instrumentation code, leaving only the overhead associated with protecting the machine state and executing the instrumentation task itself. A series of tests was conducted on the SPEC CPU2000 Integer benchmarks in order to ascertain the validity of this optimization, which showed that for an instrumentation tool that counts the number of basic block executions in the application run, the overhead is reduced by 45% by inlining the instrumentation.

IV. Results

To investigate the efficiency of instrumented executables created by PEBIL, we ran several experiments on the SPEC CPU2000 Integer benchmark suite using the reference input set. All of the experiments are run on a single core of a quad-core 2.4GHz IA32 Intel Xeon running Red Hat Linux Enterprise 4.1.2 (Linux kernel 2.6.18) and each runtime presented is the average runtime of three identical runs unless otherwise noted.

The first set of experiments quantifies the overhead of the program relocation and transformation techniques used by PEBIL as described in Section III-A. Recall that PEBIL modifies the structure of the ELF executable, adds an additional unconditional branch execution to each function call in order to relocate the function then extends all of the branches in the code to use 32-bit offsets. In order to measure the

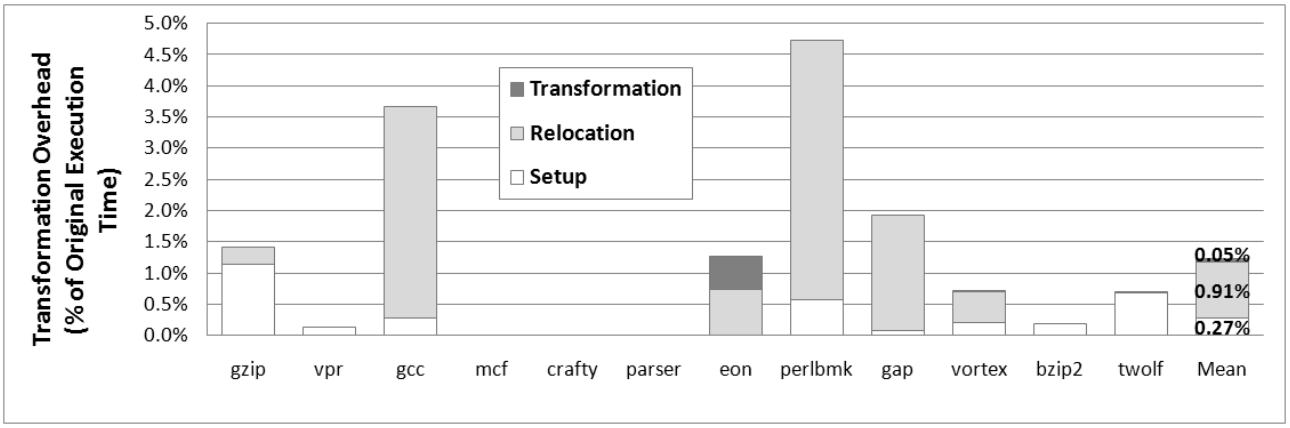


Fig. 4. Application overhead caused by preparing the code for instrumentation but without any instrumentation inserted.

overhead caused by these steps: (1) setup, (2) relocation and (3) transformation, these techniques were applied in stages so that the effect of each stage can be examined.

Figure 4 presents the runtime overhead of each stage as it accumulates onto the previous stages for the SPEC CPU2000 Integer benchmarks. Modifying the structure of the ELF executable has an overhead that averages 0.27% of the original application runtime, with a maximum penalty of 1.14%. Relocating the functions of a program carries an overhead that averages 0.91%, with a maximum overhead of 4.17%. Finally, converting all of the branches to use 32-bit offsets has an overhead that averages 0.05% with a maximum penalty of 0.53%. In total, the penalty incurred by preparing the executables for instrumentation has an overhead that averages 1.23%, with a maximum overhead of 4.73%. Among a set of popular dynamic instrumentation toolkits, Pin, DynamoRIO, and Valgrind, the lowest overhead for running the application within the instrumentation tool but performing no instrumentation is obtained by using DynamoRIO, which has an average of 38% overhead and a maximum overhead of 113% [1]. These results show that the overhead involved with the instrumentation preparation techniques used by PEBIL does not undermine the goal of producing efficient instrumented code.

The next set of experiments measure the overhead introduced by instrumenting the application to count the number of times each basic block is executed. We use this particular instrumentation tool because basic block counting is a case where we expect PEBIL to generate efficient instrumented executables since the number of instrumentation points required is high and the amount of work done at each instrumentation point is low, thus highlighting the overhead introduced by the tool. Much of the work performed in basic block counting, namely updating a single counter every time a basic block is encountered, can be done easily using an instrumentation snippet rather than by a full instrumentation function. The counters embodied in the instrumentation snippets must also be persistent throughout the entire run of the application, which is more suited to a static instrumentation approach because static instrumentation does not utilize any resources at runtime to determine whether instrumentation should be removed.

Figure 5 presents the overhead introduced by a basic block counter as a percentage of the original application runtime. The data presented for Pin, DynamoRIO and Valgrind is from previously published research [1]. There, the `eon` benchmark is excluded from the results because DynamoRIO was unable to instrument `eon` at the time. The data presented for Dyninst uses the static instrumentation feature of Dyninst 6.1.

The overhead of PEBIL's basic block counter ranges from 28%-111% with an average overhead of 62%. The average overhead is 96% for Dyninst's static instrumentation toolkit (ranging from 41%-179%), 151% for Pin (ranging from 8%-350%), 408% for DynamoRIO (ranging from 58%-693%), 734% for Valgrind (ranging from 91%-1483%). Our experiments demonstrate that executables instrumented by PEBIL run with an average of 35% less overhead than Dyninst and 59% less overhead than Pin, which is 34% and 89% of the original application runtimes respectively.

The instrumentation provided by Dyninst's static instrumentation toolkit is, in many respects, similar to that provided by PEBIL. We suspect that the difference in performance has two major causes. The first is that Dyninst uses a function relocation mechanism wherein each branch in the relocated function points to the branch target at the correct address in the *original* application code, which subsequently points to the correct branch target in the relocated application code. This results in the execution of two control transfers in the instrumented application for every taken branch in the original. Furthermore Dyninst uses the `pushf`/`popf` instructions in order to save and restore the contents of the flags register at each instrumentation point when protection of its contents are necessary. For basic block counting PEBIL uses the `lahf` and `sahf` instructions to accomplish this task, which are less likely to cause pipeline stalls than `pushf`/`popf` since they operate on fewer bits of the `flags` register.

V. Future Work

PEBIL currently does not perform any register analysis when considering how to insert instrumentation. Like Pin and Dyninst, PEBIL could perform liveness analysis on the individual registers and on the individual bits of the `flags` register. By doing the former PEBIL could first allocate registers from the set of dead registers in order to avoid saving and restoring their contents around an instrumentation point. By doing the latter PEBIL would be able to choose the most

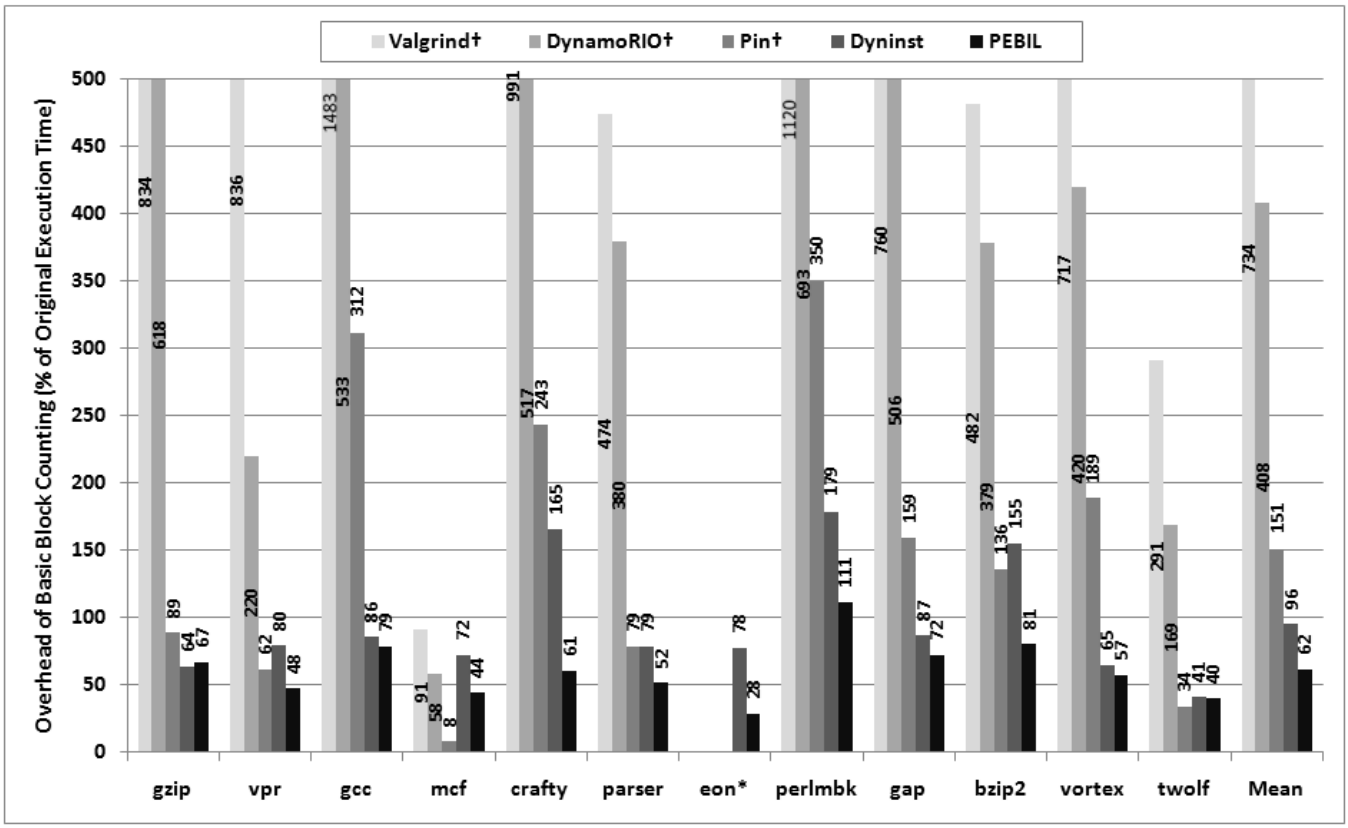


Fig. 5. Performance of several x86 instrumentation tools with basic block counting instrumentation.

efficient method of `flags` protection while still keeping its contents safe. This would take the form of not protecting the `flags` register in any way if all the flags bits are dead, protecting only the subset of the bits handled by `lahf`/`sahf` when those bits are the only live ones, or protecting all of the bits when necessary using `pushf`/`popf`.

Currently PEBIL allows the tool writer to specify which mechanism is used, and it is therefore up to the tool writer to determine which mechanism is safe to use based on what he knows about the instrumentation code being inserted. Generally this is limited to the tool writer's knowledge of the instrumentation code he is inserting rather than information about the context into which the instrumentation code is being inserted. Automatic analysis of both the instrumentation code and the application code can serve to further reduce the amount of time spent unnecessarily saving and restoring machine state, and can be implemented in a relatively straightforward manner using tools like XED, VEX or instructionAPI, which are the instruction decoders used by Pin, Valgrind, and Dyninst respectively.

VI. Related Work

ATOM [15] and EEL [16] are two of the earliest implementations of binary instrumentation toolkits. Both work in a way that is conceptually similar to PEBIL; instrumentation is performed on the compiled binary prior to runtime, meaning that any overhead due to code analysis and code generation is incurred outside of the instrumented application's run cycle. Unfortunately, ATOM is available only for the Alpha platform. Since this processor is not being produced anymore, ATOM is no longer viable as a long-term solution for those who wish to perform static, efficient instrumentation. EEL is another early example of a static binary instrumentation toolkit. EEL was designed to provide a platform-independent interface for editing executables so that it would require minimal understanding of the underlying architecture in order to make changes to the executable. This design philosophy differs from the philosophy used when designing PEBIL. PEBIL exposes certain platform-dependent details to the tool writer so that the tool writer has the potential to use his knowledge of the underlying platform to get performance benefits for the instrumented application.

BIRD [14] is a binary rewriting platform for Windows/x86. BIRD is similar to PEBIL because it uses what they call a *redirecting* approach to executing instrumentation code, which is another way of saying that they insert a 5-byte branch at the instrumentation point. BIRD uses some clever techniques to avoid it, but when 5 bytes are not available it relies on a software interrupt in order to get control from the application to instrumentation. Software interrupts are generally very heavyweight; PEBIL instead uses function relocation so that the 5-byte branch can be used at all instrumentation points.

† Results are derived from previously published research [1].

∗ Results for `eon` are not included in [1] because DynamoRIO was unable to instrument it.

Dyninst [2] is a popular static and dynamic instrumentation toolkit, meaning that it can operate either as a runtime instrumentation engine or by writing a persistent instrumented binary to disk. Similar in concept to what is done in PEBIL and BIRD, in order to accomplish a control transfer to the instrumentation code Dyninst replaces an instruction from the application with a branch instruction. Dyninst employs several optimizations that currently are not available in PEBIL, most notably that they perform liveness analysis on the registers and individual bits of the flags register. It is our plan to integrate this type of analysis into PEBIL in the future.

Pin [1] is another popular dynamic binary instrumentation toolkit that uses a JIT-based (Just In Time compilation) approach to instrumentation. This approach entails running the application on top of Pin, while Pin intercepts the application at its natural control flow interruptions so that it can instrument upcoming parts of the program. For efficiency, Pin performs many optimizations including caching these instrumented sequences of code to allow for re-use, chaining instrumented sequences of code together to avoid unnecessary tool intervention, and avoiding state protection overheads whenever possible.

DynamoRIO [4] and Valgrind [3] are two other dynamic binary instrumentation toolkits that use a JIT-based approach to instrumentation and operate in a similar fashion to Pin. These toolkits offer certain functionality that is not available anywhere else. Valgrind offers support for a feature called shadow values [6], which can be used to create instrumentation tools that are difficult to build without this feature. An example of this is a tool that tracks the initialization of every bit in the program's data in order to show when the program accesses uninitialized data. While incredibly useful, support of this kind entails a more heavyweight approach to instrumentation that is unsatisfactory when efficiency is the primary goal.

VII. Conclusions

In this paper we introduced an efficient static instrumentation toolkit, PEBIL, for Linux on the x86/x86_64 platform family. PEBIL uses function relocation in order to subsequently transform the application code so that it can create enough space to enable fast instrumentation at arbitrary points in an executable. PEBIL's function relocation mechanism and the ability to insert efficient assembly code snippets allow PEBIL to produce efficient instrumented executables. The overhead of PEBIL for counting basic block executions on a set of applications is 65% of the overhead of Dyninst, 41% of the overhead of Pin, 15% of the overhead of DynamoRIO, and 8% of the overhead of Valgrind. PEBIL is freely available to the public for download at [13].

VIII. Acknowledgements

This work was supported in part by the DOD High Performance Computing Modernization Program, the DOE Office of Science through the SciDAC2 award entitled Performance Evaluation Research Center and the DOE through the High Performance Computing Research Program.

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] R.A. Uhlig and T.N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2), 1997.

[6] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM New York, NY, USA, 2007.

[7] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7. Citeseer, 1997.

[8] B.P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes.

[9] A. Snavely, X. Gao, C. Lee, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance modeling of HPC applications. *Dresden, Germany, August*, 2003.

[10] 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.

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

[12] 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.

[13] Performance Modeling and Characterization: PEBIL Project Homepage. http://www.sdsc.edu/pmac/projects/pebil.html.

[14] S. Nanda, W. Li, L.C. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370. IEEE Computer Society Washington, DC, USA, 2006.

[15] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205. ACM New York, NY, USA, 1994.

[16] J.R. Larus and E. Schnarr. EEL: Machine-independent Executable Editing. *ACM Sigplan Notices*, 30(6):291–300, 1995.