# Obtaining Dynamic Program Information with Binary Instrumentation

Michael Laurenzano

Department of Computer Science and Engineering
University of California, San Diego
michaell@cs.ucsd.edu

## Abstract

*Dynamic information about a program has many uses, such as aiding the programmer in debugging or optimizing his code, helping make code more secure, or helping hardware and systems designers make organizational or tradeoff decisions. Program information is available through a variety of approaches, including visual inspection of the code or its output, compiler output, hardware counters, debuggers, and program instrumentation through a number of techniques. This work contains a survey of several approaches for obtaining program information and uses the findings to argue that Binary Instrumentation allows efficient access to information that is not available from any other technique. This paper examines a series of binary instrumentation toolkits that are able to instrument a static executable or a running executable, through code patching and Just-In-Time (JIT) compilation. Here is also discussed the various advantages of these tools over one another in terms of efficiency, transparency, and portability.*

## 1 Introduction

Dynamic information about a program, such as memory access patterns and how efficiently the code is running, has many uses over the lifetime of a program. The programmer can use dynamic program information to help debug the program, correct logical or semantic errors [Law97] [ADS93] and to tune and optimize the program [LDM$^+$01] [SMAB01] [ABL97]. Moreover, compilers can use dynamic program information to create a more efficient version of the code [CL99] [FOK02] [CFE97]. The security and correctness of the code can even be verified and tested using dynamic program information [MCI$^+$01] [SGK] [CMS01]. Outside of the program's life cycle, computer architects and operating system writers can also use this information to help guide them in making decisions about hardware and operating system design [LCM97] [SPHC02] [JL01] [KSD$^+$97].

There are several available techniques to gather information about the dynamic behavior of a program. Simple techniques include looking at source or assembly code of the program, or compiling it to determine what code is generated. More complex techniques include using hardware counters, simulating code, running a debugger on the code, or inserting instrumentation into the code. Of all these techniques, software instrumentation is one of the more useful because it provides detailed and high-level program information as well as a high degree of flexibility with respect to the type of information that is gathered.

Furthermore, among the possible ways to perform instrumentation (source level, assembly level, or binary level), binary instrumentation is the most useful strategy because it allows efficient and transparent access to some information about the program that is not available from any other technique. This leads to a detailed review of various binary instrumentation techniques: static instrumentation, dynamic JIT compilation, and dynamic code patching are techniques that can be used to instrument an executable. This paper describes the mechanics of a number of binary instrumentation tools and makes some comparisons about how these tools and techniques rate in the areas of efficiency, transparency, and portability.

The remainder of this paper is organized as follows: Section 2 discusses techniques for obtaining program information and why program instrumentation is necessary. Section 3 provides an overview of ways in which program information can be useful. Section 4 provides an overview of program instrumentation and makes the case that binary instrumentation provides information that cannot be found through other instrumentation techniques. Section 5 examines binary instrumentation, discussing specific tools and implementation issues as well as tradeoffs that must be made in tool design. Section 6 provides some concluding remarks. Appendix A provides definitions of some key terms used throughout this paper.

## 2 Obtaining Program Information

There are many ways of obtaining information about the dynamic behavior of a program. These techniques range from simply running the code and examining the output or looking at the source code to more complex techniques like counting events with hardware counters, simulating the program or inserting instrumentation.

Running the code, while easy to do, provides very little insight into the dynamic behavior of the program. One can try to reconstruct program events by examining the contents and the order of the program's output in order to get a rough idea of the values of variables and the control flow. However, such techniques are ad-hoc and typically do not provide much information. A popular technique used by novice programmers is to manually add source code to print information about the program's state so that more detailed information is given about the program's behavior during the program run. This technique can be considered a rudimentary and informal form of source code instrumentation, yet it can provide valuable insight into the rough operation of relatively simple programs. Thus it is mainly used for debugging programs. It is fairly obvious that it is infeasible to use this technique to analyze larger programs or to obtain a large amount of state information needs to be examined.

Another technique that can provide detailed information about dynamic program behavior is to simulate the program [BA97]. Simulation can provide a very high level of detail about the program, but has several inherent limitations. The simulation environment may not accurately reflect the underlying system since some details are either unknown or elided for the sake of simplicity or efficiency, meaning that the information learned about the program may not be entirely accurate. For example if one wanted to collect the memory addresses of a program, the simulation environment would have to manage memory in the same way as the underlying system. More importantly, accurate program simulation is typically extremely time consuming. Faithful simulation of each execution cycle can slow the program down by as much as a factor of $1,000$ or even $10,000$ [BA97], which renders it impractical for all but the smallest programs.

Debuggers can also be used to get information about a program [Law97] [ADS93]. While debuggers are very useful for debugging, in general they are not intended to gather whole program information because they are specifically geared towards interactive identification of bugs. This means that their standard usage requires a great deal of user interaction, where not only is the user slow, but the debugger is often not written to produce efficient code. Debuggers generally use hardware trap instructions to perform their core functions such as inserting breakpoints into an executable. Debugging is thus sometimes seen as an option for instrumentation because simple code can be inserted into the executable at almost any point.

Hardware counters can also be used to get information about what has happened during a program run [LDM$^+$01] [SMAB01] [ABL97]. In order to use hardware counters, the programmer is generally expected to either manually or automatically insert calls to a collection routine. A collection routine typically accesses the hardware counters, stores or outputs the values, and resets the counters. Since these collection routines must be inserted into the program in an automatic way, hardware counters often rely on instrumentation for their collection. Even though hardware counters gather information efficiently, they are known to have problems reporting accurate results under all conditions [sun] [DLM$^+$]. This is because they are often implemented in a fairly ad-hoc manner for their use in debugging and evaluation by the original hardware developers. Hardware counters are one of the less intrusive ways to get program information since they often do not require software support for maintaining state information and because they can be quickly accessed. However, *accessing* hardware counters does require software support at the system level. Because hardware counters are often enabled by program instrumentation of some form, these can be viewed partially as a type of performance measurement that is enabled by instrumentation and partly as an information gathering technique in their own right. While hardware counters are useful for counting program events, most of them cannot give information about program flow[1], such as address traces or control flow.

In addition to these well-known techniques to help gather dynamic program information, there are many complete instrumentation toolkits available that allow the user to insert arbitrary code into a program in order to observe its runtime behavior. The other methods already mentioned above have drawbacks; either some type of information is not available or it is computationally prohibitive to get the information. This leaves the need for a set of tools that are powerful, flexible, and delicate enough to get many types of program information transparently and efficiently. Program instrumentation meets all of these requirements. It is powerful because it allows access to the details of the program's state at almost any time during runtime. It is flexible because it allows selective choosing of events to observe and actions for those observations.

[1]For example, the Performance Monitoring Control (PMC) registers on the Itanium 2 can be used to gather information about instruction, address, and branch traces [ita].

## 3 Using Program Information

Program information is used by hardware and system designers in order to make design decisions. For example, an architect designing a branch predictor would typically evaluate the branch predictor by gathering a branch outcome trace, then simulating these branch outcomes on the predictor [JL01] [LCM97]. Similarly, one could use the memory address trace of an application to evaluate cache hardware [UM97] [SWC01]. System designers can use information such as the program profile from a benchmark to aid in the comparison of memory management techniques, file systems [PBL90], network protocols [Kes88], or thread scheduling policies.

Program information can also be used by a compiler in order to better optimize a program. In [CL99], the authors use information from a small profiled program run in order to guide the compiler to perform optimizations that are of the most benefit to the program. This profile, which is an execution count of the basic blocks in the program, is used by the compiler to expend more resources optimizing "hot" parts of the code while ignoring "cold" parts of the code. In [FOK02] a different approach is taken; the code is run iteratively while exploring the compilation space in order to find the best set of optimizations. This technique only involves running the code, and thus does not require any complex techniques to get program information. Such an approach works well on its own, but the addition of fine-grained program execution data could improve optimization or guide the optimizer more quickly through the optimization space. The work done in [CFE97] keeps track of dynamic variable values so that they can be identified as being semi-invariant. When a variable is identified as semi-invariant, the compiler can perform previously unavailable optimizations such as constant folding, code specialization and partial evaluation on that variable. This type of analysis lends itself well to binary or assembly instrumentation since it requires access to all of the program's variables, even the ones not explicitly used in the source code.

Programmers are able to use program information in a variety of ways. Programmers often need runtime information, such as control flow paths and program state, in order to aid in the debugging process [Law97] [ADS93]. In general, a debugger needs to have some way of inserting code into an application for breakpoints. Normally hardware trap instructions are used, but it can also be done using dynamic binary instrumentation [BH00], a technique that allows additional code to be inserted into the executable at runtime. After program correctness is ensured, program information can be useful to the programmer in tuning and optimizing the program. Much effort has been put into simply organizing and displaying program information to smooth the programmers optimization effort [LDM+01] [SMAB01]. Such tools rely heavily on instrumentation, both in its pure form and to access hardware counters, in order to provide the user with such information.

Dynamic program behavior can be used to test and verify the security of a program. In [SGK], instrumentation is used to fence memory buffers by surrounding them with unwritable memory, as well as inserting code to recover from the resulting segmentation faults that that occur when these unwritable locations are accessed. In this way, buffer overflow attacks would be detected instead of causing damage or unwanted intrusion into the program. Instrumentation is also used in [CMS01] to ensure that untrusted java applications run "safe" versions of particular functions in order to mitigate the risk presented by some of the standard java libraries functions. An example of this is to provide and execute a safe version of the Frame constructor of a java GUI class that does not allow more than a few windows to be opened by any particular application. This prevents an unknown application from hogging the resources of the host system. Taking a different approach to the matter of security, the authors of [MCI+01] use a dynamic instrumentation framework to demonstrate that certain types of attacks are feasible. As such, they execute a malicious process on a remote server to take control of a machine running a vulnerable distributed scheduling system, leave malicious processes there, and migrate the malicious processes to other machines. They also use instrumentation to learn where licensing checks are made by the software, then replace those calls so that the licensing requirements are bypassed.

## 4 Program Instrumentation Techniques

In the examples given in Section 3, all of the uses of program information can be satisfied by gathering information through program instrumentation. But there are many stages during the compilation of a program where instrumentation can be inserted, as shown in Figure 1 [LB94]. Several of these stages, such as the compiler, assembler, or linker, perform instrumentation by modification of existing pieces of software. Inserting instrumentation at any of these stages can be unacceptable for several reasons. If the software is commercial, the source code may not be available and thus cannot be modified. Modifying a single compiler does not help instrument programs written for a different language/compiler or programs written in multiple different languages. Since code generation occurs at the end of compilation, instrumentation would likely occur on an intermediate representation of the code so that various optimizations could be performed on the combined code. This would create at least some differences between the code being measured and the original code, making the instrumentation non-transparent. For transparency, the instrumentation should be inserted after code generation or prior to code generation but after all optimizations. Modifying the code during compilation requires re-compilation/assembling/linking the program in order to perform instrumentation. For these reasons, the modification of any existing software tools is not considered in any more depth. Instead, binary instrumentation is the topic of focus[2].

[2]Instrumentation of assembled (but not linked) objects is similar in many respects to instrumenting the linked executable, thus it will not be treated as being any different from binary instrumentation.

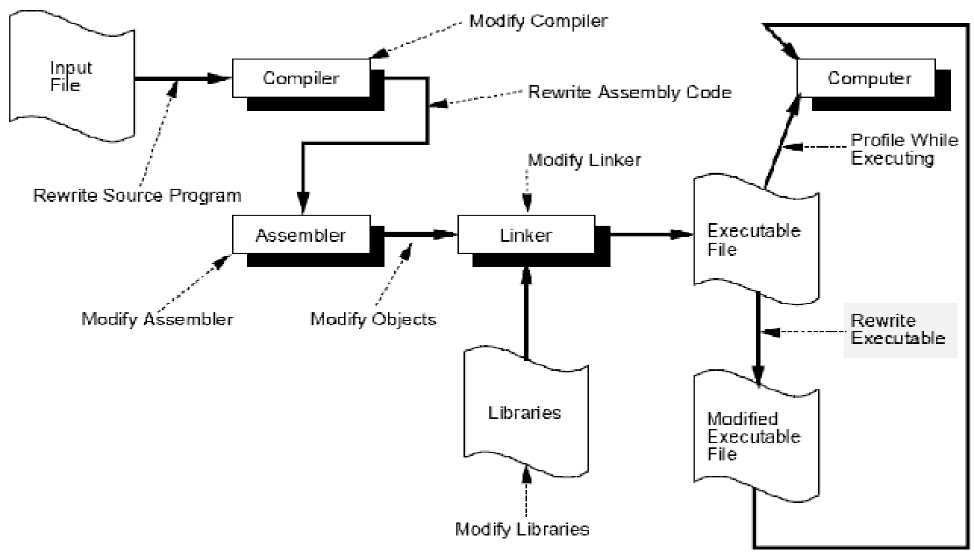

Figure 1: Locations within the program creation cycle where instrumentation can be inserted.

## 4.1 Source Code Instrumentation

In source level instrumentation, instrumentation is inserted into the source code of a program. In general, since the control structure and other interesting events in the source code must be recognized by the instrumentation system, a source level instrumentation system will be tied at least partly to the language in which the program is written. Instrumentation can be aware of and occur around high-level code features, something that is not possible when using lower levels of instrumentation [TJ98]. Source instrumentation also requires access to the source code and recompilation of the program.

Source code is also easier to instrument compared to other stages in the compilation process since the compiler, not the instrumentation tool writer, will deal with the complexity involved in mixing the instrumentation and original code. There have been attempts to use source level instrumentation as the basis for performance tuning [Yan94]. But since the compiler typically treats any inserted code as part of the original program's code, these two bodies of code become convoluted. This means that instrumentation is unlikely to be transparent. For example, inserting a call to an instrumentation routine could prevent the compiler from performing a code motion optimization through that point that would have been performed in the absence of the instrumentation routine. Such instrumentation may produce inaccurate information, especially if the instrumentation is being used to help optimize the code. This is because the original parts of the instrumented version of the code will be optimized far differently than those same parts in the version of the code that is the target of the optimization. For this reason source level instrumentation is generally used to aid in tuning and optimization only by gathering course-grained program information [SMAB01] [LDM+01] or for the debugging of applications with the insertion of print statements.

Towards making source level instrumentation more transparent and powerful, there has been some work involving instrumentation-aware compilation [She01]. The compiler strips the code of instrumentation, compiles the code while maintaining a table that indicates how optimizations affect the removed instrumentation, then reinserts the instrumentation with the aid of this table after the code has been optimized. Since this technique requires significant changes to the compiler design it has not been widely used in practice.

Source level instrumentation is also lacking in that particular details of program execution cannot be accessed. For example, a common use of instrumentation is to capture memory addresses accessed during program execution. The compiled code often contains many loads and stores that cannot be directly mapped to a variable usage in the program's source code. Probably the most common cause of these extra memory references is register spilling, where the number of registers in the ISA is fewer than the number of registers in the working set of the application. This disparity occurs between the source code and the assembly code, and is one of the major drawbacks of source level instrumentation.

One application of source level instrumentation is Aspect Oriented Programming (AOP) [KLM$^+$96], a programming methodology which allows the programmer to modularize program-wide concerns. An example of such a concern is logging of high-level program activity. This is a concern that could affect all the modules of a program, and is easier for the programmer to manage if the particulars such as detail level and output target for the logging can be managed as a single entity. On the other hand, AOP systems can be seen as a vehicle for performing source level instrumentation. This is the approach taken in [MSSP], where an aspect-oriented language (AspectC++) is used as the basis for implementing a program monitoring and debugging scheme.

Overall source level instrumentation is useful because it is easy to implement and has access to high-level program structure. Moreover, it tends to be fairly efficient since the compiler's optimization engine is available for optimizing both the additional code and the mixing of the original/additional code. But due to the limitations of source level instrumentation, more useful information may not be gathered. Instrumentation with respect to program measurement is not transparent is most cases, making it unsuitable for use in program optimization. There is also a disparity in what is executed on the machine and what is seen in the source code, making it unsuitable for collecting any details that are specific to the compiled code, such as memory access or control flow.

## 4.2 Assembly Code Instrumentation

Assembly level instrumentation is the act of inserting extra code into the compiled assembly code, before assembling and linking. In performing instrumentation at this stage and later stages of the compilation process, the instrumentation tool or user must deal with optimizing the areas where additional code is inserted. Assembly code is more difficult to instrument than source code since an assembly code instrumentation system must deal explicitly with register usage and platform-specific features such as function calls and indirect control flow. But it does allow some more difficult issues such as branch target resolution to be handled by the linker. The fact that the instrumentor must deal with these lower-level details also means that they are exposed for observation.

One of the earliest examples of an assembly instrumentation tool was the Software Instruction Counter (SIC) [MCL89], which counts the number of instructions executed in the program. At every basic block, SIC inserts code to increment a counter by the number of instructions found in that basic block. This example elucidates the fact that even such a simple instrumentation tool could not be created with source instrumentation because the number of instructions created by the compiler is not known in the source code and it is difficult to identify basic blocks from the source code alone. Other program details that are not exposed in the source code are exposed in the assembly code as well. In the Aliter instrumentation tool [GSS05], instrumentation is used to collect the memory address stream of an application and then to simulate these addresses in a cache simulator to evaluate theoretical memory systems. The complete address stream of the application is not available at higher levels of instrumentation, but is available in the assembly code.

However, assembly level instrumentation has several drawbacks. Assembly pseudo-ops may be generated instead of actual instructions, making the assembly code non-representative of the machine code. Some assemblers also perform optimizations on the assembly code, again introducing a disparity between the assembly code and the code that actually runs. Finally, using the compiler to generate assembly code instead of using the full compiler chain can have an effect on the optimizations used by the compiler.

## 5 Binary Instrumentation

In binary instrumentation, extra code is inserted into the program at some point after the compilation process. Since binary instrumentation tools deal directly with the compiled executable, a more detailed representation of program information can be gathered than with instrumentation tools that deal with the program on a higher level. But since executable formats are typically complex, modifying them is an error-prone process that must be handled delicately. Since instrumentation code inserted into the program will typically require registers, it is often the case that the instrumentation tool must save machine state. If the machine state is not kept identical to the original run, the instrumentation will result in incorrect execution.

The efficiency of the code produced by a binary instrumentation tool is degraded because of four major factors [GSS05]. Instrumentation causes control interruptions, requires that machine state be saved, causes more instructions be executed in the instrumented code, and pollutes the cache behavior of the original program. For the most part, the latter two factors cannot be controlled by the instrumentation tool. They are a function of the type and location of the inserted instrumentation which in general is selected by the user. To some degree, the first two factors can be controlled by the instrumentation tool. Dynamic binary instrumentation toolkits must also incur the cost of performing instrumentation at runtime. This section examines binary instrumentation toolkits in detail. Section 5.1 examines static instrumentation, where the code is modified after the compilation process but prior to execution. Then section 5.2 looks at dynamic instrumentation, where the code is modified during execution. Throughout this section, binary instrumentation schemes are evaluated in terms of transparency, efficiency and portability.

### 5.1 Static Instrumentation

A static binary instrumentation tool typically follows a certain pattern of behavior. First, the entire executable is read and the code structures, such as functions, basic blocks and instructions are identified. Then the instrumentation tool inserts additional code that is defined according to the user's specifications. The tool must ensure that the executable is left in a consistent state when it has finished modification so that the original execution behavior is retained. For example, if an instruction is inserted into the code between a branch and its target, the instrumentation tool is responsible for ensuring that the address in the branch is updated to reflect the change caused by the code insertion.

There are several ways to add instrumentation code statically. One technique is to inline the additional code at the instrumentation points where it would normally be executed. Using this technique is better for efficiency of the instrumented code because it reduces the number of branches that must be executed at each instrumentation point. Another option for inserting code is to use code patching, as is done in PMaCinst [TLCS]. Recall from Appendix A that in code patching, a single instruction in the application is replaced by a branch to some additional code. This additional code can either contain some instructions that comprise the instrumentation or some wrapper code that will branch to the instrumentation itself. Using this technique as opposed to code insertion has three major consequences. In general, it is less efficient because it requires at least two more branches (original code to instrumentation and vice-versa) than code inlining requires. It also decreases the difficulty of inserting instrumentation for two reasons. When performing code insertion, the instrumentation tool must modify every branch that targets a location after the instrumentation point since the locations of all these instructions are modified by the insertion. And when inserting additional code between existing instructions, it also becomes possible that a branch can extend beyond the distance allowed by the branch instruction. This means that either multiple branches must be used in a chain, different instructions must be used, or some form of indirect branching has to be used. Thirdly, since code insertion move code, the instrumentation tool must keep track of original addresses and instructions so that this information can be passed to the instrumentation code in a transparent fashion.

The first class of binary instrumentation tools created were simple profiling tools, such as Pixie [Smi]. Pixie is implemented on MIPS platforms and provides the ability to insert code that maintains and updates counters for each basic block and for each conditional branch. After instrumented program execution, these counters are combined with static information about the executable and with simulation to provide powerful profiling information. An example of this is to combine basic block execution counts with the instruction mix of each basic block (known statically) in order to produce the instruction mix of the program.

Another tool that is similar to Pixie is the qpt [LB94] profiler. qpt is similar to pixie in that it is a basic block counter, but it also is capable of collecting a memory address trace of an application. qpt makes a significant effort to address the concerns of portability, efficiency, and transparency. To address portability, the authors of qpt implemented their tools for both MIPS and SPARC platforms. Since modifying executables is largely platform-dependent, only some parts of the instrumentation tool can be used for both. This is the common approach taken with any portable instrumentation tools. Since a large part of any binary instrumentation tool is invariably platform-specific, parts of the code can be classified as either platform-specific and platform-independent. Then, in order to port the instrumentation toolkit to another platform, only the platform-specific parts must be ported. The platform-specific parts are typically significant, making portability in the truest sense of the word an elusive goal in binary instrumentation [LS95] [BH00] [LCM$^+$05]. qpt also addresses issues of efficiency. The authors analyze the CFG of the application in order to find its maximal weight spanning tree. By instrumenting only the blocks that comprise the maximal weight spanning tree of the application, they minimize the number of instrumentation points and thus minimize the slowdown due to instrumentation. Finally, qpt was one of the first tools that addressed some other issues of transparency. Since some indirect branch locations are difficult to detect statically, a mapping of the original indirect branch target address to the new target address is maintained at the exact location of the branch target in the program's original text section. Thus the address space for the original text section remains intact and contains these mappings, while the new code resides in a different part of the program's address space.

After these early single-purpose instrumentation tools, several general purpose instrumentation toolkits were developed. These toolkits allow the user to insert customized instrumentation at arbitrary points in the program, while they would take care of some amount of the underlying complexity of code insertion. Two early such tools were the Executable Editing Library (EEL) [LS95] and ATOM [SE94]. EEL is derived the qpt toolset. This was an attempt to provide a platform independent way of editing executables. As stated above, true platform independence is impossible since the executables themselves have many details that are tied to both the underlying system and architecture. EEL provides classes that represent program abstractions and ways of editing these abstractions; for each system, it requires a definition of how the instructions in the system can be translated into intermediate instructions. EEL then offers ways in which the program can be viewed and modified as intermediate instructions, which are then translated back into machine instructions when editing is complete. The main contribution of this work is that it attempted to modularize the machine-dependant and machine-independent parts of the code to the highest degree possible. This allows for a minimal amount of effort to be put forth when implementing EEL on a different platform.

ATOM is a static binary instrumentation toolkit on the Alpha AXP platform for OSF/1. In ATOM, the user provides an instrumentation file that tells ATOM where in the program to insert the analysis functions and an analysis file that implements the analysis functions. ATOM provides a set of classes and functions that allow the user to navigate the program, access particular program objects, and query the content of those objects such as to the various fields of an instruction (source registers, target registers, branch target, opcode, etc.). ATOM combines the user's instrumentation code and some generic object modifying code provided by ATOM in order to create a customized instrumentation tool. This tool is then applied to the application, which can be run to produce some side effects as a result of the instrumentation that is attached to the code, typically writing some information about the application to an output file. An outline of how ATOM instruments an application can be seen in Figure 2. In ATOM, extra code inserted amidst the rest of the application's text section through inlining. If the user wishes to use the PC of some instruction as an argument to an analysis routine, ATOM statically knows the original address and thus supplies the original instead of the modified address. Also, any extra variables used by the analysis routines are allocated in a space that is separate from the rest of the application's data so that the application's data addresses are not disturbed. One caveat to ATOM's transparency is that function calls can be taken indirectly, thus their locations are unknown statically. Since the user would access these function addresses by simply asking for the contents of a general purpose register holding the address, ATOM does not modify these values to correctly reflect the change to the function's location.

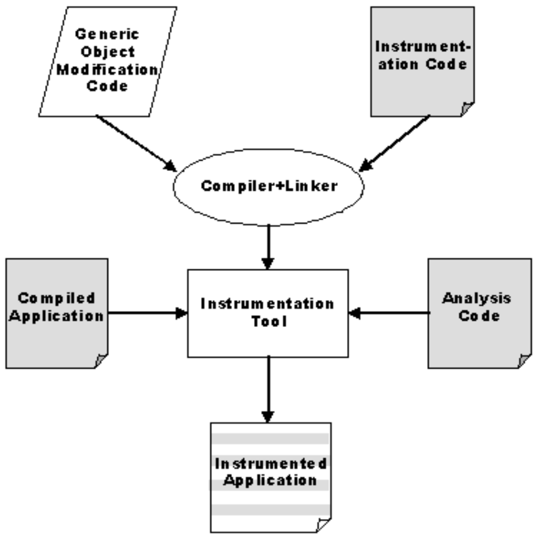

Figure 2: The stages of how an application is instrumented with ATOM.

ATOM also takes several measures that increase the efficiency of the running instrumented executable. On the Alpha/AXP platform, some registers are defined as callee-saved registers that are preserved across procedure calls and some are defined as caller-saved registers that are not preserved across procedure calls. The result of this is that any function inserted by ATOM must save any caller-saved registers. These registers are either saved by wrapper code for the function or directly inside the function. ATOM inserts one wrapper code for each function so that a copy of the code is not needed for each function invocation and this code is put at the end of the text section. Either technique lessens the amount of code that has to be added to the application since it is only added once per function instead of once per function invocation. Placing the register saving/restoration in the function's code also reduces the number of branches that must be taken at every function call from 4 to 2. ATOM also avoid saving registers that are unused in the analysis function or registers that are dead upon reaching the function. Selectively saving registers results in far less overhead per function call and greatly improves performance.

Etch [RVL+97] is a binary instrumentation platform for Win32 executables running on the x86 platform, a platform which produces a set of challenges that are different from all of the other instrumentation frameworks that have been examined so far. Code discovery on the x86 platform is more difficult because it uses variable-length instructions and executables can mix code and data (such as jump tables) together. Even discovering which modules (dynamically linked libraries, or DLLs) are used by an application requires running the program since there a large number of them, many of which are identified and loaded at runtime. Finally, Win32 applications tend to be much more sensitive to the environment in which they run, meaning that any instrumentation will have to maintain the same environment or at least the appearance of the same environment. Etch also provided leaps in usability that were not previously available. It provides a GUI that allows users to select modules and standard instrumentations by "pointing and clicking," which allows far less sophisticated users to access binary instrumentation, as well as providing a GUI that allows unsophisticated users to navigate the program data. Typically, accessing program information in this manner is done by 3rd party software (see Section 3).

## 5.2 Dynamic Instrumentation

In dynamic instrumentation, code is analyzed and instrumented at runtime. A result of this is that static instrumentation is generally more efficient with respect to its effect on the running code than dynamic instrumentation. This is because the cost of instrumentation, such as performing code discovery and generating code, must be born at runtime instead offline. However, dynamic instrumentation provides many features that are unavailable in static instrumentation. In static instrumentation, it is typically difficult to instrument dynamically loaded libraries or shared libraries because the locations and identities of these libraries are unknown until runtime. For example, Etch solved this problem by running the application once prior to instrumentation in order to discover which libraries are loaded by the application, but this technique is not fool-proof since changing the input or environment of the application can change the set of libraries used by the application. Dynamic instrumentation schemes are able to instrument shared libraries since they have access to the code that is actually being run by the program, after load time.

Dynamic instrumentation is also useful because instrumentation can be inserted and removed from the program while it is running. For example, this may be useful for measuring the performance of a production database that cannot be instrumented and restarted. Measuring this application can be done by attaching instrumentation for a period of time to get measurements then detaching instrumentation when finished. Similarly, other large applications for which full instrumentation is impractical can be instrumented for short periods of time in this manner. There are two major approaches that have been taken to dynamic binary instrumentation: JIT compilation and code-patching.

### 5.2.1 JIT Compilation

In JIT compilation the application runs on top of the instrumentation tool. In order to insert instrumentation, the instrumentation tool periodically takes control away from the application in order to instrument the upcoming code. This generally is more robust than either static instrumentation or dynamic instrumentation with code patching because it can properly handle mixed code and data, dynamically generated code, dynamically loaded libraries, and statically unavailable indirect jump targets [LCM$^+$05].

The most influential of these type of instrumentation tools is called Pin [LCM$^+$05]. Pin is implemented for 3 Intel platforms: Itanium, x86, and ARM. Pin takes control of the application and loads itself into the application's address space using the Unix ptrace library [ptr]. Pin then instruments the upcoming code for the application in a way that is specified by the user's instrumentation code and analysis routines similar to the instrumentation and analysis routines in ATOM. To regain control from the application, Pin replaces the upcoming branch in the application so that control is returned to Pin after the application executes the code currently being instrumented. While simple in concept, this basic model of Pin is somewhat inefficient so Pin must employ many optimizations.

The simplest optimization that Pin uses is to cache the instrumented portions of code so that the most commonly executed pieces of code do not need to be re-instrumented each time they are executed. To aid with this storage Pin maintains a software code cache. Also, these instrumented sequences of code can be linked together by replacing the branch to Pin with a branch to the next sequence of code if it is known. Since the target of the trace may not be known statically in the case of an indirect branch, there is also a mechanism in place to predict the target of this branch. Pin inlines some analysis procedures to mitigate the cost of register spilling and control transfers typically associated with function calls. There are also some platform-specific optimizations that are performed to aid with efficiency. The most effective of this class of optimization involves tracking the liveness of all bits in the eflags register[3] and not performing a costly save/restore around functions when every bit in this register is dead at a function call. Since Pin has access to the original application code, it is able to maintain transparency by presenting addresses and other application data from the original code rather than the modified code.

Two other tools, DynamoRIO [BA] and Valgrind [NS03] are other instrumentation tools that are based on JIT compilation. Both fundamentally operate in a similar manner to Pin with one important difference. They both use the LD_PRELOAD environment variable to load the tool into memory prior to running the application. This technique cannot work with statically linked binaries since such binaries do not perform any dynamic library loading. This also causes the addresses in the other libraries loaded by the executable to be shifted with respect to the original execution, making some of the information learned from instrumentation less than transparent. Similar to Pin, both of these tools perform basic optimizations of code caching and trace linking in order to provide more efficient execution. Valgrind is the least efficient of the three tools, followed by DynamoRIO, while Pin is the most efficient due to the high levels of optimization undertaken [LCM$^+$05].

### 5.2.2 Dynamic Code Patching

Code patching can be used for dynamic instrumentation, where the instrumentation is inserted by overwriting an instruction in the application's code with a branch to the instrumentation code. Moreover, instrumentation code may include additional code that handles register saving and restoring for the registers used in instrumentation code. A consequence of overwriting an instruction is that on architectures with variable-length instructions, instructions that are shorter than a branch cannot be overwritten. This requires that multiple instructions be overwritten in order to implement such a branch. Code is not inlined using this technique. Thus instrumentation overhead may be significant depending on the instrumentation code inserted and number of instrumentation points. An obvious result of less efficient instrumented code created by performing dynamic code patching is that fine-grained instrumentation on larger applications results in significant overhead.

The most well known instrumentation toolkit that uses code patching is DyninstAPI [BH00]. DyninstAPI is implemented for x86, SPARC, Alpha, and IBM platforms. In DyninstAPI, control of the application is garnered by using the unix ptrace library or /procfs/ file system. Dyninst loads two arrays into the address space of the application, one for instrumentation variables and the other for instrumentation code. In order to insert instrumentation, it modifies one of more instructions in the application in order to branch to a base trampoline. The base trampoline includes wrapper code that saves and restores the machine state of the program. This wrapper contains a branch instruction into the actual instrumentation code (called a snippet).The base trampoline also includes the original replaced instruction(s) from the original code. In this model of instrumentation there are at least 4 control transfers whenever instrumentation code is executed, which is for the case when inserted code is not a function from a shared library. Furthermore, adding instrumentation with Dyninst on some platforms currently causes all registers to be spilled at each instrumentation point[4], which results in inefficient instrumented code. The DyninstAPI is very portable; an instrumentation tool can be used on any architecture that implements Dyninst. Dyninst uses a high-level language to describe the instrumentation code then translate these high level instructions to the particular host architecture's machine instructions. The abstractions that are used to describe and manipulate program components are also platform-independent.

Another dynamic binary instrumentation tool that uses code-patching is the Detours toolkit [HB99]. This tool allows only for course-grained instrumentation at function calls, replacing a function call with a call to some instrumentation function while maintaining the ability to call the original functions from

[3]The bits of the eflags register on Intel architectures are used for storing condition codes.

[4]The authors of the DyninstAPI claim to be actively working on this problem.

the inserted code. This has several advantages over a detailed instrumentor. Since instrumentation is only inserted at the function level, the calling conventions of the architecture are used to preserve the state of the application. This makes such an instrumentor much easier to implement, while keeping the bloating of the code due to instrumentation to a minimum.

## 6 Conclusions

Information about the dynamic behavior of a program has uses in application, hardware, and system development. Programmers can use this information for debugging and optimization. Compilers can use this information to make better optimization decisions and produce more efficient code. Hardware and system developers can use this information to aid in making design decisions.

And while there are many ways to get information about the dynamic behavior of a program, software instrumentation has proven to be useful because it allows access to details of a program's behavior at a relatively low performance cost. Furthermore, instrumenting the compiled binary allows access to machine-specific details that are not available by instrumenting the software at any other level.

In any binary instrumentation tool, care must be taken to ensure that the information presented by the instrumentation to the user accurately reflects what occurred in the application. This concept is called transparency, and often times the instrumentation tool must maintain information about the contents of the original program so that it is available in the modified program [SE94] [LCM$^+$05] [LS95]. If transparency is not maintained, the instrumentation may be misleading and lose its value.

Binary instrumentation tools are inherently linked to particular host architectures. The best efforts at portability have been the result of separation of software into platform dependent and platform independent parts [LCM$^+$05] [BH00] [LS95]. Then in order to port the instrumentation to another platform, only the platform-independent parts must be implemented. This separation of concerns helps with portability, but the platform-dependant parts of the instrumentation tool make up a significant portion. See Table 1. In EEL, the platform-dependent code for SPARC and MIPS comprises 4179 of the 13960 lines of source code. In Pin, the platform-dependent code for the ARM, IA-32, and Itanium comprise 61201 lines of the 114796 lines of source code.

| Code Type | EEL | Pin |
|---|---|---|
| | # of Lines (% of Total Lines) | |
| All | 13960 (100%) | 114796 (100%) |
| Platform Ind. | 9781 (70%) | 53595 (47%) |
| Platform Dep. | 4179 (30%) | 61201 (53%) |
| Avg. Per Platform | 2090 (15%) | 20400 (18%) |

Table 1: Distribution of platform dependent and platform independent code in portable instrumentation tools.

Table 1 shows that the amount of code required to required to port EEL to another platform is much less than the effort that would be required for Pin. This is because each individual instrumentation tool implemented in EEL is responsible for providing the application with transparent and correct instrumentation code. This is the case because EEL is a more general-purpose object modification framework that can also be used for instrumentation. In Pin, which is purely an instrumentation tool, these facilities are provided in the generic instrumentation code for use by each tool. This means that any degree of true portability has escaped binary instrumentation tools. This is reasonable because these tools are working with code that is heavily tied to a machine and writing these tools in a more generic fashion is likely to negatively effect the performance of the instrumented code.

Efficiency is another chief concern of any instrumentation platform, since the utility of a tool is proportional to its performance. Much work is done to aid with the efficiency of instrumented code. In all binary instrumentation toolkits, inlining the instrumentation code and performing data flow analysis can avoid control flow and register spilling hazards respectively. In dynamic instrumentation tools, lazy code discovery and code reuse are used to speed up the instrumented code. In JIT-based dynamic instrumentation, caching instrumented code and trace linking have become standard optimizations that make this process more efficient. Many other platform-specific optimizations can also be made by a tool in order to aid with the efficiency of the code it produces. Work done in [GLSS05] shows the effects of instrumentation tool overhead incurred by the application when each memory reference is instrumented with code that stores the memory address to a global array. This measures the cost of the control flow interruptions plus the cost of saving machine state for each of the instrumentation tools. Pin, a dynamic binary instrumentation tool based on JIT compilation, is shown to create code that runs less efficiently than code created with ATOM by a factor of 1.57. This is likely to be caused by the cost of code discovery and code generation that must be born at runtime by a JIT compilation-based instrumentor despite the efficient code that is created. Finally, the code created by Dyninst is less efficient than the code created by ATOM by a factor of 15.54 and the code created by Pin by a factor of 9.88. This is probably due to the costs of instrumentation that are taken at runtime as well as the large control interruptions and state saving that occurs in Dyninst instrumentation.

In most cases, static binary instrumentation based on code inlining is the most efficient form of binary instrumentation because the costs of instrumentation are born outside of the program's execution and because inlining reduces the number of control hazards imposed by instrumentation. However, shared libraries and dynamically generated code cannot be instrumented using this method. Dynamic instrumentation tends to be the most robust because code discovery is deferred until rntime. This allows easier handling of shared libraries, dynamically generated code, and indirect jumps. Basing instrumentation on code patching is more easily implemented because the original application code is only modified by overwriting single or a few consecutive instructions in order to branch to instrumentation code. This eliminates some of the difficulty of inserting instrumentation code that must be taken by code inlining such as resolving modified branches and maintaining address transparency for the instrumentation code.

## Acknowledgements

I would like to thank Allan Snavely, whose guidance helped define the scope of this paper. He and Mustafa Tikir also helped significantly by editing early drafts of the paper.

## Appendix A: Definitions of Key Terms

**Instrumentation** - the act of inserting extra code into an application in order to observe the application's behavior.

**Static Binary Instrumentation** - instrumentation that is inserted into the compiled and linked executable prior to runtime.

**Dynamic Binary Instrumentation** - instrumentation that is inserted into a running executable.

**JIT Compilation** - in this context, JIT compilation is a dynamic binary instrumentation technique where code is translated from an uninstrumented version to an instrumented version at runtime.

**Code Patching** - an instrumentation technique where the instrumentation is enabled by replacing instructions in the code with branches to additional code.

**Transparency** - a measure of how much effect the measurement technique has on the property being measured.

**Wrapper Code** - a piece of code that is inserted along with the instrumentation that saves and restores registers to ensure that the instrumentation does not damage the application's machine state.

## References

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Notes*, 32(5):85–96, 1997.

[ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. In *Software - Practice & Experience*, June 1993.

[BA] Derek Bruening and Saman Amarasinghe. DynamoRIO: An Infrastructure for Runtime Code Manipulation.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *International Journal on High Performance Computing Applications*, 14(4):317–329, 2000.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 259–269, 1997.

[CL99] Robert Cohn and P. Geoffrey Lowney. Feedback Directed Optimization in Compaqs Compilation Tools for Alpha. *Proceedings of the 2nd Workshop on Feedback Directed Optimization*, pages 1–10, 1999.

[CMS01] Ajay Chander, John C. Mitchell, and Insik Shin. Mobile Code Security by Java Bytecode Instrumentation. *2001 DARPA Information Survivability Conference & Exposition (DISCEX II)*, 2001.

[DLM$^+$] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, Daniel Terpstra, Haihang You, and Min Zhou. Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters.

[FOK02] Grigori Fursin, Michael O'Boyle, and Peter Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.

[GLSS05] Xiaofeng Gao, Michael Laurenzano, Beth Simon, and Allan Snavely. Reducing overheads for acquiring dynamic memory traces. *IEEE International Symposium on Workload Characterization*, 2005.

[GSS05] Xiaofeng Gao, Beth Simon, and Allan Snavely. Aliter: an asynchronous lightweight instrumentation tool for event recording. *SIGARCH Computer Architecture News*, 33(5):33–38, 2005.

[HB99] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions, 1999.

[ita] Intel itanium 2 processor reference manual for software development and optimization. Website.

[JL01] Daniel A. Jimenez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 197–206, 2001.

[Kes88] Srinivasan Keshav. *Real: a Network Simulator*. University of California, Berkeley, Computer Science Division, 1988.

[KLM$^+$96] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[KSD$^+$97] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing Operating Systems using Robustness Benchmarks. *The Sixteenth Symposium on Reliable Distributed Systems.*, pages 72–79, 1997.

[Law97] Rob Law. An overview of debugging tools. *SIGSOFT Software Engineering Notes*, 22(2):43–47, 1997.

[LB94] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software - Practice and Experience*, 24(2):197–218, 1994.

[LCM97] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The bi-mode branch predictor. *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 4–13, 1997.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[LDM$^+$01] Kevin London, Jack Dongarra, Shirley Moore, Phil Mucci, Keith Seymour, and Thomas Spencer. End-user Tools for Application Performance Analysis Using Hardware Counters. *International Conference on Parallel and Distributed Computing Systems*, 2001.

[LS95] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation*, pages 291–300, New York, NY, USA, 1995. ACM Press.

[MCI$^+$01] Barton P. Miller, Mihai Christodorescu, Robert Iverson, Tevfik Kosar, Alexander Mirgorodskii, and Florentina Popovici. Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes. *Parallel Processing Letters*, 11(2/3):267–280, 2001.

[MCL89] John M. Mellor-Crummey and Thomas J. LeBlanc. A software instruction counter. *SIGARCH Computer Architecture News*, 17(2):78–86, 1989.

[MSSP] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schroder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.*

[NS03] Nichloas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, 2003.

[PBL90] Arvin Park, Jeffrey C. Becker, and Richard J. Lipton. IOStone: a synthetic file system benchmark. *ACM SIGARCH Computer Architecture News*, 18(2):45–52, 1990.

[ptr] Linux/unix command: ptrace. Website. `http://linux.about.com/library/cmd/blcmdl2_ptrace.htm`.

[RVL$^+$97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hand Levy, and Brian Bershad. Instrumentation and optimization of win32/intel executables using etch. In *USENIX Windows NT Workshop*, 1997.

[SE94] Amitabh Srivastava and Alan 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, New York, NY, USA, 1994. ACM Press.

[SGK] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. *Lecture Notes in Computer Science*, pages 1–15.

[She01] Sameer Shende. *The Role of Instrumentation and Mapping in Performance Measurement.* PhD thesis, University of Oregon, 2001.

[SMAB01] Sameer Shende, Allen D. Malony, and Robert Ansell-Bell. Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA2001*, pages 1150–1156, 2001.

[Smi] Michael D. Smith. Tracing with Pixie. Technical report, Technical Report CSL-TR-91-497, Stanford University, November 1991.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[sun] Sun studio 11: Performance analyzer readme. Website. `https://developers.sun.com/sunstudio/documentation/ss11/mr/READMEs/analyzer.html`.

[SWC01] Allan Snavely, Nicole Wolter, and Laura Carrington. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. *Proceedings of the IEEE Workshop on Workload Characterization*, 2001.

[TJ98] Kevin Templer and Clinton L. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 249–258, 1998.

[TLCS] Mustafa Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely. PMaC Binary Instrumentation Library for PowerPC/AIX.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven Memory Simulation: a Survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[Yan94] Jerry C. Yan. Performance Tuning with AIMS-An Automated Instrumentation and Monitoring System for Multicomputers. *Proceedings of the 27th Hawaii International Conference on System Sciences*, 1994.