An API for Runtime Code Patching

Bryan Buck Jeffrey K. Hollingsworth†
Computer Science Department
University of Maryland
College Park, MD 20742 USA
{buck,hollings}@cs.umd.edu

Abstract

We present a post-compiler program manipulation tool called Dyninst which provides a C++ class library for program instrumentation. Using this library, it is possible to instrument and modify application programs during execution. A unique feature of this library is that it permits machine-independent binary instrumentation programs to be written. We describe the interface that a tool sees when using this library. We also discuss three simple tools built using this interface: a utility to count the number of times a function is called, a program to capture the output of an already running program to a file, and an implementation of conditional break points. For the conditional breakpoint example, we show that by using our interface compared with gdb we are able to execute a program with conditional breakpoints up to 900 times faster.

1. Introduction

The normal cycle of developing a program is to edit source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing, and not have to re-compile, re-link, or even re-execute the program to change the binary. At first thought, this may seem like a bizarre goal, however there are several practical reasons we may wish to have such a system. For example, if we are measuring the performance of a program and discover a performance bottleneck, it might be necessary to insert additional instrumentation into the program to understand the problem. Another application is performance steering; for large simulations, computational scientists often find it advantageous to be able to make modifications to the code and data while the simulation is executing.

This paper introduces an Application Program Interface (API) to permit the insertion of code into a running program. Using this API, a program can attach to a running program, create a new bit of code and insert it into the program. The program being modified is able to continue execution and doesn't need to be re-compiled, re-linked, or even re-started. The next time the modified program executes the block of code that has been modified, the new code is executed in addition to the original code. The API also permits changing subroutine calls or removing them from the application program.

Runtime code changes are useful to support a variety of applications including debugging, performance monitoring, and supporting the composition of applications out of existing packages. Depending on the use, the code can either augment the existing program with ancillary operations such as measuring the application performance or adding additional print statements, or alternatively, it can be used to alter the semantics of the program by changing the subroutines executed or manipulating application data structures. The second type of change is most useful for either performance steering, or other debugging applications. Our API is designed to support both of these uses.

Our approach differs from other post-compiler instrumentation tools such as EEL[11], ATOM[15], or Etch[14] that permit code to be inserted into a binary before it starts to execute. Often times, the specific code to be inserted may not be known until runtime. If the user is unsure what type of instrumentation they will require, they have only two alternatives. First, they could include all possible

† Corresponding author.

instrumentation that might be required. This approach ensures that the correct information is collected, however it might be at such a high cost that it will distort or mask the relevant phenomenon to be measured. Second, they could only insert the minimum amount of instrumentation absolutely required. However, if a key bit of instrumentation is omitted, the programmer is forced to re-execute the program to enable that code. For short running programs, re-execution is not a problem. However, for long running applications such as large scientific simulations this could require hours or even days of delay. In addition, some applications such as database servers can require long periods of "warm-up" before they reach a steady state. By using runtime instrumentation, these applications can be instrumented only during the desired intervals.

We seek to provide a machine independent interface to permit the creation of tools and applications that use runtime code patching. Traditionally, post compiler instrumentation tools have provided interfaces that allow machine or assembly language level code to be inserted. Instead, our interface is more analogous to a machine independent intermediate representation of the instrumentation as an abstract syntax tree. This allows the same instrumentation code to be used on different platforms. For example, consider instrumentation code to monitor the behavior of a database system (i.e. tracking commit and abort operations). The instrumentation code would be specific to the particular database system, but because the instrumentation is machine independent, it would work with any machine architecture where the database system was installed.

A key feature of this interface is that it insertion and alteration to instrumentation in a running program. The underlying work that makes this possible is the dynamic instrumentation technology[8] developed as part of the Paradyn Parallel Performance Tools project [12].

The goal of this API is to keep the interface small and easy to understand. At the same time it needs to be sufficiently expressive to be useful for a variety of applications. The way we have done this is by providing a set of abstractions for programs and a simple way to specify the code to insert into the application. To generate more complex code, extra (initially un-called) subroutines can be linked into the application program, and calls to these subroutines can be inserted at runtime via this interface. These routines can be either statically linked, or loaded at runtime as part of a dynamic library. Although this API can be used directly by programmers, it is primarily aimed at tool builders. As a result, the interface to code generation, based on ASTs, is convenient for tool builders, yet somewhat clumsy for hand construction.

The rest of this paper is divided as follows. Section 2 introduces the basic abstractions that we provide. Section 3 describes the key classes in the API. Section 4 provides an overview of how the API is implemented. Section 5 introduces a couple of applications of the API and illustrates the advantages of using it. Section 6 describes related work, and finally Section 7 contains conclusions and future work.

2. Abstractions

The API is based on abstractions of a program and its state while in execution. The two primary abstractions are points and snippets. A *point* is a location in a program where instrumentation can be inserted. A *snippet* is a representation of a bit of executable code to be inserted into a program at a point. For example, if we wished to record the number of times a procedure was invoked, the point would be the first instruction in the procedure, and the snippets would be used to create a statement to increment a counter. Snippets can include conditionals, function calls, and loops.

The API is designed so that a single instrumentation process can insert snippets into multiple processes executing on a single machine. To support multiple processes, two additional abstractions, threads and images, are included in the API. A *thread* refers to a thread of execution. Depending on the programming model, a thread can correspond to either a normal process or a lightweight thread. *Images* refer to the

static representation of a program on disk. Images contain points where their code can be modified. Each thread is associated with exactly one image.

The overall structure of the API and its implementation is shown in Figure 1. There are two processes, which we call the mutator and the application. The left side of the figure shows the code for the mutator process that contains calls into the Dyninst API. It also contains the code that implements the runtime compiler and the utility routines to manipulate the application process (shown below the rectangle labeled API). The right half of the figure shows the application process with the original code of the program shown in the top part of the figure. The bottom two parts of the application are the snippets that are inserted into the program, and the runtime library that supports the Dyninst API. Additional details about how the implementation works are given in Section 4.

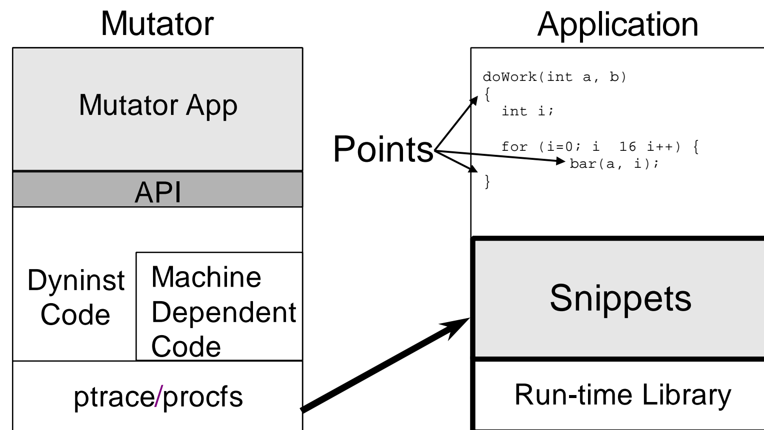

Figure 1: Abstractions Used in the API.

The API includes a simple type system to support integers, strings, and floating point values. Additionally, support for aggregate types including arrays and structures is provided. The interface allows manipulation of user defined types that exist in the target application to be modified. There is no way to create new types using the interface, but a specific tool built using the API can create new types as part of its runtime library that is loaded into the application.

An inherent part of an API to manipulate other processes is the need to react to events of interest that take place in the application process. There are two types of events that occur in an application process, events caused by the inserted code and events that occur as a result of the normal execution of the application such as process termination. To provide a uniform way to handle these events, we have defined a variety of callbacks that inform the mutator of events of interest in the application. In addition, there are functions to query if an event has happened.

3. Interface

In this section we describe the primary classes of the API, and explain their relationship to each other. There are three main components to the interface. First, there are the classes used to manipulate code in execution. This group includes the BPatch class, and the BPatch_thread class. Second, there are classes for accessing the original program and its data structures. These include the BPatch_image, BPatch_module, and BPatch_function classes. Third, there are classes to construct new code snippets and insert them. These include the BPatch_snippet class, and the BPatch_point classes.

BPatch This class represents the entire Dyninst API library. There can only be one instance of this class at a time. This class is used to perform functions and obtain information not specific to a particular

thread or image. It is also used to define the callback functions to be invoked for specific application events.

BPatch_thread operates on (and creates) code in execution. This class can be used to manipulate the thread. For example, a thread may be started, stopped, or terminated by using methods in the class. In addition, the thread class is used to insert the instrumentation code into the program. For threaded applications, the interface represents a single thread of execution. The implementation of the API ensures that even if the instrumentation code is inserted into a specific thread of a multi-threaded application that the code is only executed for that thread. For non-threaded code, the thread abstraction represents a traditional process.

BPatch_image is the abstraction that represents the program executable. Images only exist as part of threads since two processes that have the same program on disk can load during their execution different dynamic libraries, and thus have different modules available for instrumentation.

BPatch_module represents a program module, which is part of a program's executable image. Modules are provided because they are often a unit of program decomposition that is meaningful. Generally, a module refers to a single source file in the original program. However, for many libraries (especially dynamically loaded libraries), the module abstraction is used to represent the entire library.

BPatch_function represents a function in the application. A function is often a useful level of abstraction for instrumentation, and so there are methods to get the entry and exit of a function and use them as instrumentation points. In addition, there are methods of the class to determine the subroutines called by a function as well as the loops and code blocks in the function.

BPatch_point's are locations in an application's code at which the library can insert instrumentation. Points can either be described symbolically (i.e., the entry point to a function), by descending the function hierarchy (i.e. a loop), or by providing a virtual address in the program (i.e., instrumenting a specific statement or instruction).

Bpatch_type defines the interface to the type system. Types can either be pre-defined language types, or user defined types that occur somewhere in the application program. The type system is used to allocate variables for use in code snippets, and to provide a way for mutator programs or snippets to access existing application variables.

BPatch_snippet is an abstract representation of code to insert into a program. Snippets are defined by creating a new instance of the appropriate subclass of a snippet. For example, to create a snippet to call a function, a new instance of the class BPatch_funcCallExpr is created. Creating a snippet does not result in code being inserted into an application. Instead, code is generated when a request is made to insert a snippet at a specific point in a program. Sub-snippets may be shared by different snippets (i.e., a snippet may be passed as an argument to create two different snippets), but whether the generated code is shared (or replicated) between two snippets is implementation dependent. The details of the snippets are presented in the next subsection.

3.1 Code Snippets

In this section, we describe how we represent code to be generated. A collection of instances of the class BPatch_snippet, and specific sub-classes that represent different types of code to be inserted represent the statements to be added to the application by the mutator. The collection of snippets forms a direct a-cyclic graph. The code is defined by calling the appropriate C++ constructors, and passing previously created sub-snippets to each constructor. In this way, the AST is created from the leaf nodes up to the root. We now briefly describe each of the types of code snippets.

BPatch_variableExpr is derived from the snippet class. It represents a variable or area of memory in a thread's address space. A BPatch_variableExpr can be obtained from a BPatch_thread using the `malloc` member function, or from a BPatch_image using the `findVariable` member function. BPatch_variableExpr provides two member functions not provided by other types of snippets: `readValue` and `writeValue`. These methods allow the mutator program to read or write the value of a variable in the application's address space.

BPatch_arithExpr is used for most two operand statements in our code definitions. Arithmetic expressions cover a large class of operations including variable assignment, basic mathematical operations, and array references. Arithmetic operations are only supported for predefined types. For C++ programs that specify overloaded operators, the API user must directly invoke the operator function rather than using expression notation. In addition to the standard unary operations such as negation and pointer de-reference, the available binary operators are:

Operator	Description
BPatch_assign	assign the value of `rOperand` to `lOperand`
BPatch_plus	add `lOperand` and `rOperand`
BPatch_minus	subtract `rOperand` from `lOperand`
BPatch_divide	divide `rOperand` by `lOperand`
BPatch_times	multiply `rOperand` by `lOperand`
BPatch_mod	compute the remainder of dividing `rOperand` into `lOperand`
BPatch_ref	Array reference of the form `lOperand[rOperand]`
BPatch_seq	Define a sequence of two expressions (similar to comma in C)
BPatch_min	Return the smaller of two operands
BPatch_max	Return the larger of two operands.

BPatch_boolExpr expression snippets define a set of comparison operations between two variables. The operations are only defined on the base types (i.e., integer). As with BPatch_arithExpr, C++ operator overload functions must be called manually if they are to be used.

Operator	Function
BPatch_lt	Return lOperand < rOperand
BPatch_eq	Return lOperand == rOperand
BPatch_gt	Return lOperand > rOperand
BPatch_le	Return lOperand <= rOperand
BPatch_ne	Return lOperand != rOperand
BPatch_ge	Return lOperand >= rOperand
BPatch_and	Return lOperand && rOperand (Boolean and)
BPatch_or	Return lOperand || rOperand (Boolean or)

BPatch_gotoExpr provides a simple form of branching within snippets using a goto expression. The goto expression permits a snippet to branch back to an earlier part of that snippet. By combining it with conditional statements it can be used to construct loops. However, for complex loops it is generally better to write the code as a function, and use the API to patch in a call to that function.

4. Implementation

In this section, we provide a high level description of the process used to compile instrumentation code and patch programs at runtime. Although the main focus of this paper is on the high-level abstractions, a bit of information about the implementation is useful to understand the expected performance of the runtime generated code. Complete details about the implementation are available in other papers[8, 9].

While there are many tricky details in trying to shoehorn new code into running programs, one of the key features of the API is that it elides these details from the tool builder.

The basic operations on the application process by the mutator process employ the same operating system services used by debuggers (e.g., ptrace, /proc filesystem, etc.). These services provide a way to control process execution, and to read and write the address space of the application program. In addition, a dynamic linked library that contains utility functions and two large arrays is loaded into the application to be instrumented. Both arrays are used for dynamically allocating small regions of memory. One of the arrays is used for instrumentation variables, and the other to hold instrumentation code[1]. Both of these arrays are managed as heaps by the mutator process to provide dynamically allocated storage for runtime code generation.

In order to generate code, we translate the snippet into machine language code in the memory of the mutator process, and then copy it into the array in the application address space. The most difficult part of inserting instrumentation is carefully modifying the original code to branch into the newly generated code. To do this, we use short sections of code called *trampolines*. Figure 2 shows the structure of a trampoline and its relationship to the instrumentation point. Trampolines provide a way to get from the point where we wish to insert the instrumentation code to our newly generated code. To do this, we replace one or more instructions at the instrumentation point with a branch to the start of a base trampoline. The base trampoline code then branches to a mini-trampoline. The mini-trampoline saves the appropriate machine state (such as the registers and condition codes), and contains the code for a single snippet. At the end of the snippet, we place code to restore the machine state and to branch back to the base trampoline. The base trampoline then executes the instruction(s) that were displaced from the original code. If the snippet is to be inserted after the point executes (i.e., after a function call return), we can also insert a mini-trampoline here.

Multiple snippets can be inserted at a single point, and they are chained together such that the end of one snippet branches to the start of the next one, and the final snippet branches back to the trampoline.

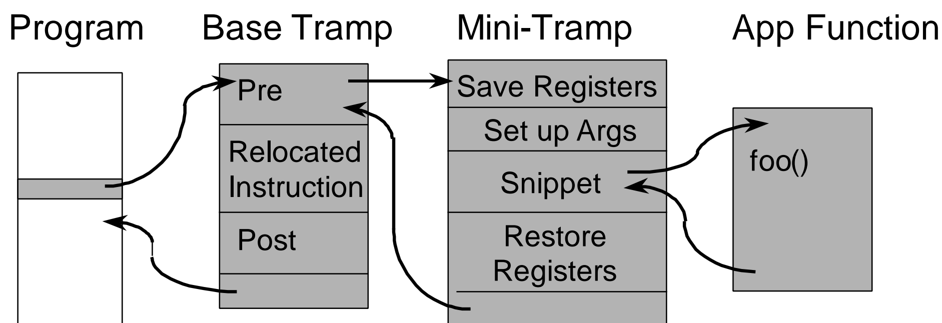

Figure 2: Inserting Code into a Running Program.

5. Using the Dyninst API

To provide insight into how our API can be used to build tools, we provide a description (and a small amount of sample code) for three applications. The first application shows a simple code snippet to increment a variable whenever a selected procedure is invoked. The second application, retee, demonstrates a simple standalone utility that can be built using the API. The third example, fast conditional break points, shows how the API could be used as part of a larger tool such as a correctness debugger.

[1] We use two separate arrays since on many platforms, instructions and data must be in separate regions of memory.

Using the API to directly create programs is possible, but somewhat tedious. We anticipate that most users of the API will be tool builders who will create higher level languages for specifying instrumentation. For example, the MDL language[9] provides a simple metric scripting language that is suited for a specific task, such as defining performance metrics. We anticipate that other "little languages" will be built to use the API for specific purposes.

5.1 Procedure Call Counting

To illustrate the ideas of the API, we first present a short example that inserts instrumentation into a target procedure to count the number of times the procedure is called. Although a trivial example, it is useful to illustrate the key features of the API.

The example code for this tool is shown in Figure 3. The first thing a mutator program must do is to create a single instance of the top-level class called BPatch. This object is used to access functions and information that are global to the library. Line 1 of the program does this.

Second, the mutator identifies the application process to be modified. If the process is already in execution, this can be done by specifying the executable file name and process id of the application as arguments to create an instance of a thread object. Alternatively, if a new process is to be created, a call can be made to the createProcess routine (as shown in line 2).

Once the application thread has been created, the mutator defines the snippet of code to be inserted and the points where they should be inserted. In our example, lines 4 and 5 show the call to lookup the handle to the entry point for our target procedure. The return value is actually a list of points since procedures may be cloned in different locations in the program, or may be overloaded. Lines 6-7 create a new integer variable in the address space of the application. The first step in creating a new variable is to lookup the type. Once a handle to the type is found, the malloc method is used to create an instance of that type. Line 8 and 9 show the process of constructing a simple increment of an integer variable. This requires constructing an integer constant expression, an addition expression, and then an assignment statement. Finally, line 10 shows the insertion of the increment statement at the desired point in the program.

```
1    Bpatch bpatch;

2    BPatch_thread *appThread->bpatch.createProcess(pathname, argv);
3    BPatch_image *appImage = appThread->getImage();

4    BPatch_Vector<BPatch_point*> *points =
5        appImage->findProcedurePoint("InterestingProcedure", BPatch_entry);

6    BPatch_variableExpr *intCounter =
7        appThread->malloc(*appImage->findType("int"));

8    BPatch_arithExpr addOne(BPatch_assign, *intCounter,
9        BPatch_arithExpr(BPatch_plus, *intCounter, BPatch_constExpr(1)));

10   appThread->insertBlock(addOne, *points);
```

Figure 3: Code to count the number of occurrences of "Interesting Procedure".

5.2 Retee Example

In this section we show an almost complete program to demonstrate the use of the API. The example is a program called "retee." We call the application "retee" since it works like the Unix command "tee," passing output to its own standard out while also saving it in a file, but unlike the "tee" command it

can be started after the application program has begun execution. The motivation for the example program is that a running program starts to print copious amount of output to the screen, and the user wishes to save that output in a file without having to re-run the program. Retee takes three arguments: the pathname of an executable program, the process id of a running instance of the same program, and a file name. It adds code to the running program that copies to the named file all output that the program writes to its standard output file descriptor.

This tool first uses the one time code feature of the API to force the application to open the file to be used for logging. The one time code feature allows the mutator to invoke a snippet once and then have control return to the mutator program. This is useful for various types of initialization code.

The code to open the file in the application is shown in Figure 4. The first line looks up the handle for the function “open”. Lines 2-8 construct the parameter list for the open call. The first parameter (lines 3-4) is a character string that is taken as input via a command line argument to the “retee” application. When the snippet containing the string is inserted into the application, the string will also be copied from the mutator to the application address space. The second and third parameters (lines 5-8) each contain an integer with file mode and protection bits. Line 9 contains a statement that constructs the overall function call from the function name and parameter list. Lines 10-11 create a new variable of type integer in the application’s address space. Line 12 constructs a statement that assigns the return value of the open system call to the new variable created in line 11. Finally, line 13 uses the one-shot code interface to compile, and execute the constructed snippet.

```
1    BPatch_function *openFunc = appImage->findFunction("open");

2    BPatch_Vector<BPatch_snippet *> openArgs;

3    BPatch_constExpr fileName(argv[3]);
4    openArgs.push_back(&fileName);
5    BPatch_constExpr fileFlags(O_WRONLY|O_CREAT);
6    openArgs.push_back(&fileFlags);
7    BPatch_constExpr fileMode(0666);
8    openArgs.push_back(&fileMode);

9    BPatch_funcCallExpr openCall(*openFunc, openArgs);

10   BPatch_variableExpr *fdVar =
11     appThread->malloc(*appImage->findType("int"));

12   BPatch_arithExpr openExpr(BPatch_assign, *fdVar, openCall);

13   appThread->OneTimeCode(openExpr);
```

Figure 4: Code to open the log file in the application.

The second part of the retee program inserts code at the entry point to the write function in the C runtime library and then checks that the write system call is for file descriptor number 1 (i.e. standard output). If the call meets this test, an additional call to the write function is made to repeat the write statement and send the output to a file we have previously opened. Note, that although our instrumentation snippet is recursively invoked, the check for the file descriptor equal to one will fail on the second call and so there is no problem with infinite recursion.

The code to generate and insert this snippet is shown in Figure 5. Line 1 locates the write system call in the application. Lines 2-7 generate a parameter list for the function call to the write system call using the created file descriptor (from Figure 4). The parameter expression is used on lines 3 and 4 to access

the buffer and length parameters of the original call to write. The statement on line 8 creates the function call itself. Lines 9-11 first create a relational expression to check that the file descriptor is zero, and then generate an if statement to call the write system call only when the conditional statement is true. Finally, line 12 inserts the constructed snippet into the application program.

```
1   BPatch_function *writeFunc = appImage->findFunction("write");

2   BPatch_Vector<BPatch_snippet *> writeArgs;
3   BPatch_paramExpr paramBuf(1);
4   BPatch_paramExpr paramNbyte(2);
5   writeArgs.push_back(fdVar);
6   writeArgs.push_back(&paramBuf);
7   writeArgs.push_back(&paramNbyte);

8   BPatch_funcCallExpr writeCall(*writeFunc, writeArgs);

9   BPatch_boolExpr compareFd(BPatch_eq, BPatch_paramExpr(0),
10                            BPatch_constExpr(1));
11  BPatch_ifExpr logStdout(compareFd, writeCall);

12  appThread->insertSnippet(logStdout, *points);
```

Figure 5: Code snippet to instrument a write system call.

5.3 Conditional Breakpoints

As a demonstration of the Dyninst API, we wrote a program that controls an application process like a debugger, and allows a user to set conditional breakpoints at any locations that can be instrumented by the Dyninst API. The user can add or remove any number of breakpoints during the running of the application. Conditional breakpoints are very expensive in most debuggers, because they are typically implemented using code that resides in the debugger rather than in the application being debugged. This means that the debugger must set an unconditional breakpoint and wait for the process to stop at it. When it does, the debugger makes system calls to look into the address space of the application and check the condition, and then automatically continues the application if the condition is not met.

Our demonstration program takes a different approach. It compiles the condition for the breakpoint into a Dyninst API code snippet (BPatch_snippet) that checks the condition and generates a signal if the condition is met. This snippet is then inserted into the application at the location where the breakpoint is desired using the `insertSnippet` member function of BPatch_thread.

Conditional breakpoints are useful, for instance, when a particular piece of code is called many times during the execution of a program, but is known or suspected to behave incorrectly only under certain conditions. Some examples might include when a function is called with certain parameters, or when the program reaches a certain point in processing its input.

The demonstration program was written in an afternoon, and consists of 371 lines of C++ code, plus 78 lines of lex specification and 149 lines of yacc, to parse the user supplied conditional expression. The whole program totals less than 600 lines of code.

Because checking of the condition is done by code that has been patched into the application, the application can be allowed to run without intervention from the debugger until the breakpoint is reached, saving the cost of potentially many context switches and system calls. To quantify how this can the degrade performance when running the application under a debugger, we ran two applications under both our tool and the GNU debugger. For each case, we measured the performance when conditional breakpoints

were inserted in various locations. The two programs were taken from the SPEC '95 benchmark suite[1]. The first, compress95, is the Unix compress program. The other, li, is a Lisp interpreter. We performed the tests on a 167 Mhz UltraSparc-1 running Solaris 2.5.

application	Breakpoints # of operations	ops/sec	dyninst time (sec)	gdb time (sec)
compress95	32,513	406,655.7	0.08	74.35
li (xlmatch)	110,209	43,607.7	2.53	221.04
li (compare)	4,475	640.2	6.99	16.39
li (binary)	401	19.4	20.69	21.62

Figure 6: Conditional Breakpoint Performance.

The results of these tests, averaged over 20 runs, are shown in Figure 6. The first column shows the application and the function where the breakpoint is inserted. The second column, "# of operations", is a count of the number of times the breakpoint was reached and the condition evaluated during the run, and "ops/sec" column shows the number of times the breakpoint condition was evaluated per second when running using our tool. The fourth column shows the wall clock time required for the breakpoint condition to become true using our tool, and "gdb time" refers to the wall clock time under gdb version 4.17.

In compress95, we inserted a breakpoint in the function "output," which outputs a code (a token representing a string of bytes). The breakpoint stops the execution of the program the first time the "output" function is called while using 16-bit codes (the size of the codes used by the program increases as the program processes the input file). A breakpoint like this might be useful, for instance, if the program was exhibiting a bug only when the code reaches a certain size. We began timing at the beginning of the "compress" function. The "ouput" function is called often, and as a result the overhead of gdb being involved in evaluating the breakpoint is extremely high, as can be seen by the fact that it took the application almost 930 times longer to reach the breakpoint for the final time.

In li, we inserted a breakpoint in one of three functions (xlmatch, compare, and binary). In each case, the breakpoint stops the program when the function is called with a certain parameter. The results show how our program's advantage over gdb decreases as the frequency that the breakpoint condition must be evaluated decreases. With the breakpoint in the least frequently called function that we tried (binary), the running time under gdb was still about 4.5% longer than that seen under breakpoint. Averaging the results of all the experiments, gdb appears to slow down the application by approximately 2 milliseconds longer than the Dyninst API program for each time the condition must be evaluated.

5.4 Other Applications of the API

The three examples above provide a flavor of the types of tools that can be built using the Dyninst API. There are several other projects underway that are making use of the API. In addition to the ones described below, a few other projects around the world are using the API, but are at too early of a stage to report here.

At the University of Maryland, we have used the API to implement an online version of critical path analysis for parallel programs running on SMPs[6]. In this application, instrumentation code to track the synchronization events in a shared-memory program is inserted on demand using the API.

At Maryland, we are also investigating using Dyninst as part of the Harmony project[7]. The Harmony project is looking into using runtime observations of applications to automatically tune programs by selecting from candidate configurations. Runtime code patching will be used to change which versions of procedures and libraries get called at specific locations. A similar use of the Dyninst interface would be to

eliminate redundant synchronization during the execution of a parallel program. For example, Diniz and Rinard[3] have proposed compiling a program to multiple different binary versions and selecting between them at runtime based on performance observations. Our API could be used to provide a mechanism to do this type of optimization.

A team at IBM is building a family of tools based on their version of the API, called DPCL[13]. These tools include a dynamic trace logger, a tool to gather statistics from hardware performance monitors, and a tool to help with understanding the performance of MPI message passing programs.

6. Related Work

One area of research that is similar to our Dyninst work is the area of binary editing tools[11, 14, 15]. Binary editing permits code to be inserted into an application binary either after it has been compiled, or sometimes after it has been linked. These tools are useful because they avoid the need to recompile the application program. However, they still require the instrumentation code to remain fixed for the lifetime of an execution of the program. One advantage of binary editing tools is that they permit more efficient code sequences to be generated because they are able to more aggressively restructure the program since it is not executing while being edited. Eel and Etch also differ from our approach in that their programming interface is at the level of either assembly language rather than a machine independent level.

One system that is designed for runtime code generation is ‘C[4]. It allows a program to define a set of statements in a C like language, and then have the program call them. This approach is a bit different than ours. First, ‘C lacks any provision to insert code at arbitrary locations in the program, and instead requires that application to explicitly call the runtime generated code. This approach is useful for things like partial evaluation and some types of runtime optimization, but is not appropriate for instrumentation or debugging.

Another common way to insert instrumentation is by having the compiler directly instrument the program as part of the compilation process. The AE[10] tool and the Convex performance tools[5] use this approach. An advantage of compiler inserted instrumentation is that the instrumentation code can be fully optimized by the compiler. However, this approach also requires that the instrumentation code remain static for the duration of the application’s execution. In addition, it of course requires re-compiling the program. For small programs, re-compilation is not a problem. However, for a large application or commercial systems re-compilation can be time consuming or the source code may not even be available.

The Los Alamos debugger [2] used a form of runtime code generation to create fast conditional breakpoints. Our approach is similar to theirs except that we provide a machine independent framework that would make it possible to write such a debugger in a portable way.

7. Summary and Conclusions

In this paper we have presented a simple API to allow runtime generation and patching of application programs. We also briefly explained how this interface is implemented. By keeping the abstractions at a machine independent level, we have been able to create portable tools. In addition, we showed three simple applications that demonstrate the API, the types of tools that can be created using it, and the potential performance improvements possible compared to alternative approaches.

We have implemented the API on Intel x86, Sun Sparc, Compaq Alpha, and IBM Power platforms. The code is freely available for research and evaluation purposes[2]. Also, the technology is being incorporated into the DPCL tool being developed by IBM[13].

[2] Please see http://www.cs.umd.edu/projects/dyninstAPI to download a copy.

8. Acknowledgements

We thank the members of the Paradyn Performance Measurement Tools project at the Universities of Wisconsin and Maryland for their useful comments about the API, and their implementation of features on a collection of different platforms. This research was supported in part by DOE Grant DE-FG02-93ER25176 and NSF awards ASC-9703212 & ASC-9711364.

9. References

1. "System Performance Evaluation Cooperative," *Capacity Management Review*, **21**(8), 1993, pp. 4-12.
2. J. S. Brown, *The Application of Code Instrumentation Technology in the Los Alamos Debugger*, Los Alamos National Laboratory, October 1992, .
3. P. Diniz and M. Rinard, "Dynamic Feedback: An Effective Technique for Adaptive Computing," *Programming Languages Design and Implementation (PLDI).* June 1997, Las Vegas, NV, pp. 71-84.
4. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek, "'C: a language for high-level, efficient, and machine-independent dynamic code generation," *POPL 96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*. Jan. 1996, St. Petersburg Beach, FL, pp. 131-44.
5. G. J. Hansen, C. A. Linthicum, and G. Brooks, "Experience with a Performance Analyzer for Multithreaded Application," *1990 International Conference on Supercomputing*. June 11-15, 1990, Amsterdam, pp. 124-131.
6. J. K. Hollingsworth, "Critical Path Profiling of Message Passing and Shared-memory Programs," *IEEE Transactions on Parallel and Distributed Computing*, **9**(10), 1998, pp. 1029-1040.
7. J. K. Hollingsworth and P. J. Keleher, "Prediction and Adaptation in Active Harmony," *The 7th International Symposium on High Performance Distributed Computing*. July 1998, Chicago, pp. 180-188.
8. J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., pp. 841-850.
9. J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation," *International Conference on Parallel Architectures and Compilation Techniques (PACT).* Nov. 1997, San Francisco, pp. 201-212.
10. J. R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software Practice and Experience*, **20**(12), 1990, pp. 1241-1258.
11. J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *PLDI*. June 18-21, 1995, La Jolla, CA,.ACM, pp. 291-300.
12. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer*, **28**(11), 1995, pp. 37-46.
13. D. M. Pase, *Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide*, IBM, June 30, 1998, http://www.ptools.org/projects/dpcl/tutref.ps.
14. T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. H. Levy, and B. Bershad, "Instrumentation and optimization of Win32/Intel executables using Etch," *Proceedings of the USENIX Windows NT Workshop*. Aug. 1997, Seattle, WA, USA, pp. 1-7.
15. A. Srivastava and A. Eustace, "ATOM: A system for Building Customized Program Analysis Tools," *SIGPLAN Conference on Programming Language Design and Implementation*. May 1994, Orlando, FL, pp. 196-205.