Paradyn Parallel Performance Tools

Dyninst Programmer's Guide

Release 10.1
May 2019

Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742
Email: dyninst-api@cs.wisc.edu

Web: www.dyninst.org
github.com/dyninst/dyninst

Contents

1 Introduction

The normal cycle of developing a program is to edit the 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 or after it has been linked, thus avoiding the process of re-compiling, re-linking, or even re-executing the program to change the binary. At first, this may seem like a bizarre goal, however, there are several practical reasons why we may wish to have such a system. For example, if we are measuring the performance of a program and discover a performance problem, 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 document describes an Application Program Interface (API) to permit the insertion of code into a computer application that is either running or on disk. The API for inserting code into a running application, called dynamic instrumentation, shares much of the same structure as the API for inserting code into an executable file or library, known as static instrumentation. The API also permits changing or removing subroutine calls from the application program. Binary code changes are useful to support a variety of applications including debugging, performance monitoring, and to support composing applications out of existing packages. The goal of this API is to provide a ma-chine independent interface to permit the creation of tools and applications that use runtime and static code patching. The API and a simple test application are described in [1]. This API is based on the idea of dynamic instrumentation described in [3].
The key features of this interface are the abilities to:

- Insert and change instrumentation in a running program.
- Insert instrumentation into a binary on disk and write a new copy of that binary back to disk.
- Perform static and dynamic analysis on binaries and processes.

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. We accomplished this goal by providing a simple set of abstractions and a way to specify which code to insert into the application . [1]

[1] 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.

2 Abstractions

The DyninstAPI library provides an interface for instrumenting and working with binaries and processes. The user writes a *mutator*, which uses the DyninstAPI library to operate on the application. The process that contains the *mutator* and DyninstAPI library is known as the *mutator process*. The *mutator process* operates on other processes or on-disk binaries, which are known as *mutatees*.

The API is based on abstractions of a program. For dynamic instrumentation, it can be based on the state while in execution. The two primary abstractions in the API are *points* and *snippets*. A *point* is a location in a program where instrumentation can be inserted. A *snippet* is a representation of some 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 entry point of the procedure, and the *snippets* would be a statement to increment a counter. *snippets* can include conditionals and function calls.

Mutatees are represented using an *address space* abstraction. For dynamic instrumentation, the *address space* represents a process and includes any dynamic libraries loaded with the process. For static instrumentation, the *address space* includes a disk executable and includes any dynamic library files on which the executable depends. The *address space* abstraction is extended by *process* and *binary* abstractions for dynamic and static instrumentation. The *process* abstraction represents information about a running process such as threads or stack state. The *binary* abstraction represents information about a binary found on disk.

The code and data represented by an *address space* is broken up into *function* and *variable* abstractions. *Functions* contain *points*, which specify locations to insert instrumentation. *Functions* also contain a *control flow graph* abstraction, which contains information about basic blocks, edges, loops, and instructions. If the *mutatees* contains debug information, DyninstAPI will also provide abstractions about *variable* and *function types*, *local variables*, *function parameters*, and *source code line inforamtion*. The collection of *functions* and *variable* in a *mutatees* is represented as an *image*.

The API includes a simple type system based on structural equivalence. If mutatee programs have been compiled with debugging symbols and the symbols are in a format that Dyninst understands, type checking is performed on code to be inserted into the mutatee. See Section 4.28 for a complete description of the type system.

Due to language constructs or compiler optimizations, it may be possible for multiple functions to *overlap* (that is, share part of the same function body) or for a single function to have multiple *entry points*. In practice, it is impossible to determine the difference between multiple overlapping functions and a single function with multiple entry points. The DyninstAPI uses a model where each function (BPatch_function object) has a single entry point, and multiple functions may overlap (share code). We guarantee that instrumentation inserted in a particular function is only executed in the context of that function, even if instrumentation is inserted into a location that exists in multiple functions.

3 Examples

To illustrate the ideas of the API, we present several short examples that demonstrate how the API can be used. The full details of the interface are presented in the next section. To prevent confusion, we refer to the application process or binary that is being modified as the mutatee, and the program that uses the API to modify the application as the mutator. The mutator is a separate process from the application process.

The examples in this section are simple code snippets, not complete programs. Appendix A - Complete Examples provides several examples of complete Dyninst programs.

3.1 Instrumenting A Function

A mutator program must create a single instance of the class `BPatch` . This object is used to access functions and information that are global to the library. It must not be destroyed until the mutator has completely finished using the library. For this example, we assume that the mutator program has declared a global variable called `bpatch` of class `BPatch` .
All instrumentation is done with a `BPatch_addressSpace` object, which allows us to write codes that work for both dynamic and static instrumentation. During initialization we use either `BPatch_process` to attach to or create a process, or `BPatch_binaryEdit` to open a file on disk. When instrumentation is completed, we will either run the `BPatch_process`, or write the `BPatch_binaryEdit` back onto the disk.

The mutator first needs to identify the application 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 in order to create an instance of a process object:

```
BPatch_process *appProc = bpatch.processAttach(name, processId);
```

If the mutator is opening a file for static binary rewriting, it executes:

```
BPatch_binaryEdit *appBin = bpatch.openBinary(pathname);
```

The above statements create either a `BPatch_process` object or `BPatch_binaryEdit` object, depending on whether Dyninst is doing dynamic or static instrumentation. The instrumentation and analysis code can be made agnostic towards static or dynamic modes by using a `BPatch_addressSpace` object. Both `BPatch_process` and `BPatch_binaryEdit` inherit from `BPatch_addressSpace`, so we can use cast operations to move between the two:

```
BPatch_process *appProc = static_cast<BPatch_process *>(appAddrSpace)
−or−
BPatch_binaryEdit *appBin = static_cast<BPatch_binaryEdit *>(appAddrSpace)
```

Similarly, all instrumentation commands can be performed on a `BPatch_addressSpace` object, allowing similar codes to be used between dynamic instrumentation and binary rewriting:

```
BPatch_addressSpace *app = appProc;
−or−
BPatch_addressSpace *app = appBin;
```

Once the address space has been created, the mutator defines the snippet of code to be inserted and identifies where the points should be inserted.

If the mutator wants to instrument the entry point of `InterestingProcedure`, it should get a `BPatch_function` from the application's `BPatch_image`, and get the entry `BPatch_point`

from that function:

```
std::vector<BPatch_function *> functions;
std::vector<BPatch_point *> *points;
BPatch_image *appImage = app->getImage();
appImage->findFunction("InterestingProcedure", functions);
points = functions[0]->findPoint(BPatch_locEntry);
```

The mutator also needs to construct the instrumentation that it will insert at the `BPatch_point`. It can do this by allocating an integer in the application to store instrumentation results, and then creating a `BPatch_snippet` to increment that integer :

```
BPatch_variableExpr *intCounter =
   app->malloc(*(appImage->findType("int")));

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

The mutator can set the `BPatch_snippet` to be run at the `BPatch_point` by executing an `insertSnippet` call:

```
app->insertSnippet(addOne, *points);
```

Finally, the mutator should either continue the mutate process and wait for it to finish, or write the resulting binary onto the disk, depending on whether it is doing dynamic or static instrumentation:

```
appProc->continueExecution();
while (!appProc->isTerminated()) {
   bpatch.waitForStatusChange();
} -or-
appBin->writeFile(newPath);
```

A complete example can be found in Appendix A - Complete Examples.

3.2 Binary Analysis

This example will illustrate how to use Dyninst to iterate over a function's control flow graph and inspect instructions. These are steps that would usually be part of a larger data flow or control flow analysis. Specifically, this example will collect every basic block in a function, iterate over them, and count the number of instructions that access memory.

Unlike the previous instrumentation example, this example will analyze a binary file on disk. Bear in mind, these techniques can also be applied when working with processes. This example makes use of InstructionAPI, details of which can be found in the InstructionAPI Reference Manual. Similar to the above example, the mutator will start by creating a BPatch object and opening a file to operate on:

```
BPatch bpatch;
BPatch_binaryEdit *binedit = bpatch.openFile(pathname);
```

The mutator needs to get a handle to a function to do analysis on. This example will look up a function by name; alternatively, it could have iterated over every function in `BPatch_image` or `BPatch_module`:

```
BPatch_image *appImage = binedit->getImage();

std::vector<BPatch_function *> funcs;
image->findFunction(\InterestingProcedure", funcs);
```

A function's control flow graph is represented by the `BPatch_flowGraph` class. The `BPatch_flowGraph` contains, among other things, a set of `BPatch_basicBlock` objects connected by `BPatch_edge` objects. This example will simply collect a list of the basic blocks in `BPatch_flowGraph` and iterate over each one:

```
BPatch_flowGraph *fg = funcs[0]->getCFG();

std::set<BPatch_basicBlock *> blocks;
fg->getAllBasicBlocks(blocks);
```

Given an `Instruction` object, which is described in the InstructionAPI Reference Manual, we can query for properties of this instruction. InstructionAPI has numerous methods for inspecting the memory accesses, registers, and other properties of an instruction. This example simply checks whether this instruction accesses memory:

```
std::set<BPatch_basicBlock *>::iterator block_iter;
for (block_iter = blocks.begin(); block_iter != blocks.end(); ++block_
iter) {
   BPatch_basicBlock *block = *block_iter;
   std::vector<Dyninst::InstructionAPI::Instruction::Ptr> insns;
   block->getInstructions(insns);
}
```

Given an `Instruction` object, which is described in the InstructionAPI Reference Manual, we can query for properties of this instruction. InstructionAPI has numerous methods for inspecting the memory accesses, registers, and other properties of an instruction. This example simply checks whether this instruction accesses memory:

```
std::vector<Dyninst::InstructionAPI::Instruction::Ptr>::iterator insn_
iter;
for (insn_iter = insns.begin(); insn_iter != insns.end(); ++insn_iter)
{
   Dyninst::InstructionAPI::Instruction::Ptr insn = *insn_iter;
   if (insn->readsMemory() || insn->writesMemory()) {
      insns_access_memory++;
   }
}
```

3.3 Instrumenting Memory Access

There are two snippets useful for memory access instrumentation: `BPatch_effectiveAddressExpr` and `BPatch_bytesAccessedExpr`. Both have nullary con-structors; the result of the snippet depends on the instrumentation point where the snippet is insert-ed. `BPatch_effectiveAddressExpr` has type void*, while `BPatch_bytesAccessedExpr` has type int.

These snippets may be used to instrument a given instrumentation point if and only if the

point has memory access information attached to it. In this release the only way to create instrumentation points that have memory access information attached is via `BPatch_function.findPoint(const std::set<BPatch_opCode>&)`. For example, to instrument all the loads and stores in a function named InterestingProcedure with a call to printf, one may write:

```
BPatch_addressSpace *app = ...;
BPatch_image *appImage = proc->getImage();

// We're interested in loads and stores
std::set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);

// Scan the function InterestingProcedure and create instrumentation points
std::vector<BPatch_function*> funcs;
appImage->findFunction(\InterestingProcedure", funcs);
std::vector<BPatch_point*>* points = funcs[0]->findPoint(axs);

// Create the printf function call snippet
std::vector<BPatch_snippet*> printfArgs;
BPatch_snippet *fmt = new BPatch_constExpr("Access at:  printfArgs.push_-
back(fmt);
BPatch_snippet *eae = new BPatch_effectiveAddressExpr();
printfArgs.push_back(eae);

// Find the printf function
std::vector<BPatch_function *> printfFuncs;
appImage->findFunction("printf", printfFuncs);

// Construct the function call snippet
BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

// Insert the snippet at the instrumentation points
app->insertSnippet(printfCall, *points);
```

4 Interface

This section describes functions in the API. The API is organized as a collection of C++ classes. The primary classes are `BPatch` , `BPatch_process`, `BPatch_binaryEdit`, `BPatch_thread`, `BPatch_image`, `BPatch_point`, and `BPatch_snippet`. The API also uses a template class called std::vector. This class is based on the Standard Template Library (STL) vector class.

4.1 CLASS BPatch

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

```
std::vector<BPatch_process*> *getProcesses()
```

Returns the list of processes that are currently defined. This list includes processes that were directly created by calling processCreate/processAttach, and indirectly by the UNIX fork or the Windows CreateProcess system call. It is up to the user to delete this vector when they are done with it.

```
BPatch_process *processAttach(const char *path, int pid, BPatch_hybrid-
Mode mode=BPatch_normalMode)
BPatch_process *processCreate(const char *path, const char *argv[], const
char **envp = NULL, int stdin_fd=0, int stdout_fd=1, int stderr_fd=2, BPatch_-
hybridMode mode=BPatch_normalMode)
```

Each of these functions returns a pointer to a new instance of the BPatch_process class. The path parameter needed by these functions should be the pathname of the executable file containing the process image. The processAttach function returns a BPatch_process associated with an existing process. On Linux platforms the path parameter can be NULL since the executable image can be derived from the process pid. Attaching to a process puts it into the stopped state. The processCreate function creates a new process and returns a new BPatch_process associated with it. The new process is put into a stopped state be-fore executing any code.

The stdin_fd, stdout_fd, and stderr_fd parameters are used to set the standard input, output, and error of the child process. The default values of these parameters leave the input, output, and error to be the same as the mutator process. To change these values, an open UNIX file descriptor (see open(1)) can be passed.

The `mode` parameter is used to select the desired level of code analysis. Activating hybrid code analysis causes Dyninst to augment its static analysis of the code with run-time code discovery techniques. There are three modes: `BPatch_normalMode`, `BPatch_-exploratoryMode`, and `BPatch_defensiveMode`. Normal mode enables the regular static analysis features of Dyninst. Exploratory mode and defensive mode enable addtional dynamic features to correctly analyze programs that contain uncommon code patterns, such as malware. Exploratory mode is primarily oriented towards analyzing dynamic control transfers, while defensive mode additionally aims to tackle code obfuscation and self-modifying code. Both of these modes are still experimental and should be used with caution. Defensive mode is only supported on Windows.

Defensive mode has been tested on normal binaries (binaries that run correctly under normal mode), as well as some simple, packed executables (self-decrypting or decompress-

ing). More advanced forms of code obfuscation, such as self-modifying code, have not been tested recently. The traditional Dyninst interface may be used for instrumentation of binaries in defensive mode, but in the case of highly obfuscated code, this interface may prove to be ineffective due to the lack of a complete view of control flow at any given point. Therefore, defensive mode also includes a set of callbacks that enables instrumentation to be performed as new code is discovered. Due to the fact that recent efforts have focused on simpler forms of obfuscation, these callbacks have not been tested in detail. The next release of Dyninst will target more advanced uses of defensive mode.

`BPatch_binaryEdit *openBinary(const char *path, bool openDependencies = false)`

This function opens the executable file or library file pointed to by path for binary rewriting. If `openDependencies` is true then Dyninst will also open all shared libraries that path depends on. Upon success, this function returns a new instance of a `BPatch_binaryEdit` class that represents the opened file and any dependent shared libraries. This function re-turns NULL in the event of an error.

`bool pollForStatusChange()`

This is useful for a mutator that needs to periodically check on the status of its managed threads and does not want to check each process individually. It returns true if there has been a change in the status of one or more threads that has not yet been reported by either `isStopped` or `isTerminated`.

`void setDebugParsing (bool state)`

Turn on or off the parsing of debugger information. By default, the debugger information (produced by the –g compiler option) is parsed on those platforms that support it. However, for some applications this information can be quite large. To disable parsing this information, call this method with a value of `false` prior to creating a process.

`bool parseDebugInfo()`

Return true if debugger information parsing is enabled, or `false` otherwise.

`void setTrampRecursive (bool state)`

Turn on or off trampoline recursion. By default, any snippets invoked while another snippet is active will not be executed. This is the safest behavior, since recursively-calling snippets can cause a program to take up all available system resources and die. For example, adding instrumentation code to the start of printf, and then calling printf from that snippet will result in infinite recursion.

This protection operates at the granularity of an instrumentation point. When snippets are first inserted at a point, this flag determines whether code will be created with recursion protection. Changing the flag is ***not*** retroactive, and inserting more snippets will not change the recursion protection of the point. Recursion protection increases the overhead of instrumentation points, so if there is no way for the snippets to call themselves, calling

this method with the parameter `true` will result in a performance gain. The default value of this flag is `false`.

`bool isTrampRecursive ()`

Return whether trampoline recursion is enabled or not. `True` means that it is enabled.

`void setTypeChecking(bool state)`

Turn on or off type-checking of snippets. By default type-checking is turned on, and an attempt to create a snippet that contains type conflicts will fail. Any snippet expressions created with type-checking off have the type of their left operand. Turning type-checking off, creating a snippet, and then turning type-checking back on is similar to the type cast operation in the C programming language.

`bool isTypeChecked()`

Return `true` if type-checking of snippets is enabled, or `false` otherwise.

`bool waitForStatusChange()`

This function waits until there is a status change to some thread that has not yet been re-ported by either isStopped or isTerminated, and then returns true. It is more efficient to call this function than to call pollForStatusChange in a loop, because waitForStatusChange blocks the mutator process while waiting.

`void setDelayedParsing (bool)`

Turn on or off delayed parsing. When it is activated Dyninst will initially parse only the symbol table information in any new modules loaded by the program, and will postpone more thorough analysis (instrumentation point analysis, variable analysis, and discovery of new functions in stripped binaries). This analysis will automatically occur when the infor-mation is necessary.

Users which require small run-time perturbation of a program should not delay parsing; the overhead for analysis may occur at unexpected times if it is triggered by internal Dyninst behavior. Users who desire instrumentation of a small number of functions will benefit from delayed parsing.

`bool delayedParsingOn()`

Return `true` if delayed parsing is enabled, or `false` otherwise.

`void setInstrStackFrames(bool)`

Turn on and off stack frames in instrumentation. When on, Dyninst will create stack frames around instrumentation. A stack frame allows Dyninst or other tools to walk

a call stack through instrumentation, but introduces overhead to instrumentation. The default is to not create stack frames.

`bool getInstrStackFrames()`

Return `true` if instrumentation will create stack frames, or `false` otherwise.

`void setMergeTramp (bool)`

Turn on or off inlined tramps. Setting this value to `true` will make each base trampoline have all of its mini-trampolines inlined within it. Using inlined mini-tramps may allow instrumentation to execute faster, but inserting and removing instrumentation may take more time. The default setting for this is `true`.

`bool isMergeTramp ()`

This returns the current status of inlined trampolines. A value of `true` indicates that trampolines are inlined.

`void setSaveFPR (bool)`

Turn on or off floating point saves. Setting this value to false means that floating point registers will never be saved, which can lead to large performance improvements. The default value is `true`. Setting this flag may cause incorrect program behavior if the instrumentation does clobber floating point registers, so it should only be used when the user is positive this will never happen.

`bool isSaveFPROn ()`

This returns the current status of the floating point saves. `True` means we are saving float-ing points based on the analysis for the given platform.

`void setBaseTrampDeletion(bool)`

If `true`, we delete the base tramp when the last corresponding minitramp is deleted. If `false`, we leave the base tramp in. The default value is `false`.

`bool baseTrampDeletion()`

Return `true` if base trampolines are set to be deleted, or `false` otherwise.

`void setLivenessAnalysis(bool)`

If `true`, we perform register liveness analysis around an instPoint before inserting instrumentation, and we only save registers that are live at that point. This can lead to faster run-time speeds, but at the expense of slower instrumentation time. The default value is

true.

`bool livenessAnalysisOn()`

Return `true` if liveness analysis is currently enabled.

`void getBPatchVersion(int &major, int &minor, int &subminor)`

Return Dyninst's version number. The major version number will be stored in major, the minor version number in minor, and the subminor version in subminor. For example, under Dyninst 5.1.0, this function will return 5 in major, 1 in minor, and 0 in subminor.

`int getNotificationFD()`

Returns a file descriptor that is suitable for inclusion in a call to select(). Dyninst will write data to this file descriptor when it to signal a state change in the process. `BPatch::pollForStatusChange` should then be called so that Dyninst can handle the state change. This is useful for applications where the user does not want to block in `BPatch::waitForStatusChange`. The file descriptor will reset when the user calls `BPatch::pollForStatusChange`.

`BPatch_type *createArray(const char *name, BPatch_type *ptr, unsigned int low, unsigned int hi)`

Create a new array type. The name of the type is name, and the type of each element is ptr. The index of the first element of the array is low, and the last is high. The standard rules of type compatibility, described in Section 4.28, are used with arrays created using this function.

`BPatch_type *createEnum(const char *name, std::vector<char *> &element-Names, std::vector<int> &elementIds)`
`BPatch_type *createEnum(const char *name, std::vector<char *> &element-Names)`

Create a new enumerated type. There are two variations of this function. The first one is used to create an enumerated type where the user specifies the identifier (int) for each element. In the second form, the system specifies the identifiers for each element. In both cases, a vector of character arrays is passed to supply the names of the elements of the enumerated type. In the first form of the function, the number of element in the element-Names and elementIds vectors must be the same, or the type will not be created and this function will return NULL. The standard rules of type compatibility, described in Section 4.28, are used with enums created using this function.

`BPatch_type *createScalar(const char *name, int size)`

Create a new scalar type. The name field is used to specify the name of the type, and the size parameter is used to specify the size in bytes of each instance of the type. No additional information about this type is supplied. The type is compatible with other

scalars with the same name and size.

```
BPatch_type *createStruct(const char *name, std::vector<char *> &field-
Names, std::vector<BPatch_type *> &fieldTypes)
```

Create a new structure type. The name of the structure is specified in the name parameter. The fieldNames and fieldTypes vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return NULL). The standard rules of type compatibility, described in Section 4.28, are used with structures created using this function. The size of the structure is the sum of the size of the elements in the fieldTypes vector.

```
BPatch_type *createTypedef(const char *name, BPatch_type *ptr)
```

Create a new type called name and having the type ptr.

```
BPatch_type *createPointer(const char *name, BPatch_type *ptr)
BPatch_type *createPointer(const char *name, BPatch_type *ptr, int size)
```

Create a new type, named name, which points to objects of type ptr. The first form creates a pointer whose size is equal to sizeof(void*)on the target platform where the mutatee is running. In the second form, the size of the pointer is the value passed in the size parameter.

```
BPatch_type *createUnion(const char *name, std::vector<char *> &field-
Names, std::vector<BPatch_type *> &fieldTypes)
```

Create a new union type. The name of the union is specified in the name parameter. The fieldNames and fieldTypes vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return NULL). The size of the union is the size of the largest element in the fieldTypes vector.

4.2 Callbacks

The following functions are intended as a way for API users to be informed when an error or significant event occurs. Each function allows a user to register a handler for an event. The return code for all callback registration functions is the address of the handler that was previously registered (which may be NULL if no handler was previously registered). For backwards compatibility reasons, some callbacks may pass a `BPatch_thread` object when a `BPatch_process` may be more appropriate. A `BPatch_thread` may be converted into a `BPatch_process` using `BPatch_thread::getProcess()`.

4.2.1 Asynchronous Callbacks

```
typedef void (*BPatchAsyncThreadEventCallback)( BPatch_process *proc, BPatch_-
thread *thread)
bool registerThreadEventCallback(BPatch_asyncEventType type, BPatchAsync-
ThreadEventCallback cb)
```

```
bool removeThreadEventCallback(BPatch_asyncEventType type, BPatch_Async-
ThreadEventCallback cb)
```

The type parameter can be either one of BPatch_threadCreateEvent or BPatch_thread-DestroyEvent. Different callbacks can be registered for different values of type.

5 Using Dyninst API with the component libraries

6 Using the API

6.1 OVERVIEW OF MAJOR STEPS

6.2 CREATING A MUTATOR PROGRAM

6.3 SETTING UP THE APPLICATION PROGRAM(MUTATEE)

6.4 RUNNING THE MUTATOR

6.5 OPTIMIZING DYNINST PERFORMANCE

6.5.1 Optimizing Mutator Performance

6.5.2 Optimizing Mutatees Performance

A Complete Examples

In this section we show two complete examples: the programs from Section 3 and a complete Dyninst program, retee.

A.1 INSTURMENTING A FUNCTION

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

// Attach, create, or open a file for rewriting
BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
```

```
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;
        case open:
            // Open the binary file and all dependencies
            handle = bpatch.openBinary(name, true);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}
// Find a point at which to insert instrumentation
std::vector<BPatch_point*>* findPoint(BPatch_addressSpace* app,
                const char* name,
                BPatch_procedureLocation loc) {
    std::vector<BPatch_function*> functions;
    std::vector<BPatch_point*>* points;

    // Scan for functions named "name"
    BPatch_image* appImage = app->getImage();
    appImage->findFunction(name, functions);
    if (functions.size() == 0) {
        fprintf(stderr, "No function %s\n", name);
        return points;
    } else if (functions.size() > 1) {
        fprintf(stderr, "More than one %s; using the first one\n", name);
    }

    // Locate the relevant points
    points = functions[0]->findPoint(loc);
    return points;
}

// Create and insert an increment snippet
bool createAndInsertSnippet(BPatch_addressSpace* app,
        std::vector<BPatch_point*>* points) {
    BPatch_image* appImage = app->getImage();

    // Create an increment snippet
    BPatch_variableExpr* intCounter =
        app->malloc(*(appImage->findType("int")), "myCounter");
    BPatch_arithExpr addOne(BPatch_assign,
                            *intCounter,
                            BPatch_arithExpr(BPatch_plus,
                                *intCounter,
                                BPatch_constExpr(1)));

    // Insert the snippet
```

```
    if (!app->insertSnippet(addOne, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

// Create and insert a printf snippet
bool createAndInsertSnippet2(BPatch_addressSpace* app,
        std::vector<BPatch_point*>* points) {
    BPatch_image* appImage = app->getImage();

    // Create the printf function call snippet
    std::vector<BPatch_snippet*> printfArgs;
    BPatch_snippet* fmt =
        new BPatch_constExpr("InterestingProcedure called %d times\n");
    printfArgs.push_back(fmt);

    BPatch_variableExpr* var = appImage->findVariable("myCounter");
if (!var) {
        fprintf(stderr, "Could not find 'myCounter' variable\n");
        return false;
    } else {
        printfArgs.push_back(var);
    }

    // Find the printf function
    std::vector<BPatch_function*> printfFuncs;
    appImage->findFunction("printf", printfFuncs);
    if (printfFuncs.size() == 0) {
        fprintf(stderr, "Could not find printf\n");
        return false;
    }

    // Construct a function call snippet
    BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

    //  Insert the snippet
    if (!app->insertSnippet(printfCall, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

void finishInstrumenting(BPatch_addressSpace* app, const char* newName)
{
    BPatch_process* appProc = dynamic_cast<BPatch_process*>(app);
    BPatch_binaryEdit* appBin = dynamic_cast<BPatch_binaryEdit*>(app);

    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
```

```
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    // Find the entry point for function InterestingProcedure
    const char* interestingFuncName = "InterestingProcedure";
    std::vector<BPatch_point*>* entryPoint =
        findPoint(app, interestingFuncName, BPatch_entry);
    if (!entryPoint || entryPoint->size() == 0) {
        fprintf(stderr, "No entry points for %s\n", interestingFuncName);
        exit(1);
    }

    // Create and insert instrumentation snippet
    if (!createAndInsertSnippet(app, entryPoint)) {
        fprintf(stderr, "createAndInsertSnippet failed\n");
        exit(1);
    }

    // Find the exit point of main
    std::vector<BPatch_point*>* exitPoint =
        findPoint(app, "main", BPatch_exit);
    if (!exitPoint || exitPoint->size() == 0) {
        fprintf(stderr, "No exit points for main\n");
        exit(1);
    }

    // Create and insert instrumentation snippet 2
    if (!createAndInsertSnippet2(app, exitPoint)) {
        fprintf(stderr, "createAndInsertSnippet2 failed\n");
        exit(1);
    }
    // Finish instrumentation
```

```
    const char* progName2 = "InterestingProgram-rewritten";
    finishInstrumenting(app, progName2);
}
```

A.2 BINARY ANALYSIS

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_function.h"
#include "BPatch_flowGraph.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;
        case open:
            // Open the binary file and all dependencies
            handle = bpatch.openBinary(name, true);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}

int binaryAnalysis(BPatch_addressSpace* app) {
    BPatch_image* appImage = app->getImage();
```

```
    int insns_access_memory = 0;

    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);

    if (functions.size() == 0) {
        fprintf(stderr, "No function InterestingProcedure\n");
        return insns_access_memory;
    } else if (functions.size() > 1) {
        fprintf(stderr, "More than one InterestingProcedure; using the first one\n");
    }

    BPatch_flowGraph* fg = functions[0]->getCFG();

    std::set<BPatch_basicBlock*> blocks;
    fg->getAllBasicBlocks(blocks);

    for (auto block_iter = blocks.begin();
            block_iter != blocks.end();
            ++block_iter) {
        BPatch_basicBlock* block = *block_iter;
        std::vector<InstructionAPI::Instruction::Ptr> insns;
        block->getInstructions(insns);

        for (auto insn_iter = insns.begin();
                insn_iter != insns.end();
                ++insn_iter) {
            InstructionAPI::Instruction::Ptr insn = *insn_iter;
            if (insn->readsMemory() || insn->writesMemory()) {
                insns_access_memory++;
            }
        }
    }

    return insns_access_memory;
}




int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
```

```
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    int memAccesses = binaryAnalysis(app);

    fprintf(stderr, "Found %d memory accesses\n", memAccesses);
}
```

A.3 INSTRUMENTING MEMORY ACCESS

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

// Attach, create, or open a file for rewriting
BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;

case open:
            // Open the binary file; do not open dependencies
            handle = bpatch.openBinary(name, false);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
```

```
    }

    return handle;
}

bool instrumentMemoryAccesses(BPatch_addressSpace* app) {
    BPatch_image* appImage = app->getImage();

    // We're interested in loads and stores
    BPatch_Set<BPatch_opCode> axs;
    axs.insert(BPatch_opLoad);
    axs.insert(BPatch_opStore);

    // Scan the function InterestingProcedure
    // and create instrumentation points
    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);
    std::vector<BPatch_point*>* points =
        functions[0]->findPoint(axs);
    if (!points) {
        fprintf(stderr, "No load/store points found\n");
        return false;
    }

    // Create the printf function call snippet
    std::vector<BPatch_snippet*> printfArgs;
    BPatch_snippet* fmt = new BPatch_constExpr("Access at: 0x%lx\n");
    printfArgs.push_back(fmt);
    BPatch_snippet* eae = new BPatch_effectiveAddressExpr();
    printfArgs.push_back(eae);

    // Find the printf function
    std::vector<BPatch_function*> printfFuncs;
    appImage->findFunction("printf", printfFuncs);
    if (printfFuncs.size() == 0) {
        fprintf(stderr, "Could not find printf\n");
        return false;
    }

    // Construct a function call snippet
    BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

    // Insert the snippet at the instrumentation points
    if (!app->insertSnippet(printfCall, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

void finishInstrumenting(BPatch_addressSpace* app, const char* newName) {
    BPatch_process* appProc = dynamic_cast<BPatch_process*>(app);
    BPatch_binaryEdit* appBin = dynamic_cast<BPatch_binaryEdit*>(app);
```

```
    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    // Instrument memory accesses
    if (!instrumentMemoryAccesses(app)) {
        fprintf(stderr, "instrumentMemoryAccesses failed\n");
        exit(1);
    }

    // Finish instrumentation
    const char* progName2 = "InterestingProgram-rewritten";
    finishInstrumenting(app, progName2);
}
```

A.4 RETEE

The final example is a program called "re-tee." It takes three arguments: the pathname of an exe-cutable 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. In this way it works like "tee," which passes output along to its own standard out while also saving it in a file. The motivation for the example program is that you run a program, and it starts to print copious lines of output to your screen, and you wish to save that output in a file without having to re-run the program.

```
#include <stdio.h>
#include <fcntl.h>
#include <vector>
#include "BPatch.h"
#include "BPatch_point.h"
#include "BPatch_process.h"
```

```
#include "BPatch_function.h"
#include "BPatch_thread.h"

/*
 * retee.C
 *
 * This program (mutator) provides an example of several facets of
 * Dyninst's behavior, and is a good basis for many Dyninst
 * mutators. We want to intercept all output from a target application
 * (the mutatee), duplicating output to a file as well as the
 * original destination (e.g., stdout).
 *
 * This mutator operates in several phases. In brief:
 * 1) Attach to the running process and get a handle (BPatch_process
 *    object)
 * 2) Get a handle for the parsed image of the mutatee for function
 *    lookup (BPatch_image object)
 * 3) Open a file for output
 *    3a) Look up the "open" function
 *    3b) Build a code snippet to call open with the file name.
 *    3c) Run that code snippet via a oneTimeCode, saving the returned
 *        file descriptor
 * 4) Write the returned file descriptor into a memory variable for
 *    mutatee-side use
 * 5) Build a snippet that copies output to the file
 *    5a) Locate the "write" library call
 *    5b) Access its parameters
 *    5c) Build a snippet calling write(fd, parameters)
 *    5d) Insert the snippet at write
 * 6) Add a hook to exit to ensure that we close the file (using
 *    a callback at exit and another oneTimeCode)
 */

void usage() {
    fprintf(stderr, "Usage: retee <process pid> <filename>\n");
    fprintf(stderr, "   note: <filename> is relative to the application pro-cess.\n")
}

// We need to use a callback, and so the things that callback requires
// are made global - this includes the file descriptor snippet (see below)
BPatch_variableExpr *fdVar = NULL;

// Before we add instrumentation, we need to open the file for
// writing. We can do this with a oneTimeCode - a piece of code run at
// a particular time, rather than at a particular location.

int openFileForWrite(BPatch_process *app, BPatch_image *appImage, char *fileName) {
    // The code to be generated is:
    // fd = open(argv[2], O_WRONLY|O_CREAT, 0666);

    // (1) Find the open function
    std::vector<BPatch_function *>openFuncs;
    appImage->findFunction("open", openFuncs);
    if (openFuncs.size() == 0) {
```

```
        fprintf(stderr, "ERROR: Unable to find function for open()\n");
        return -1;
    }

    // (2) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> openArgs;

    // (3) Create a string constant expression from argv[3]
    BPatch_constExpr fileNameExpr(fileName);

    // (4) Create two more constant expressions _WRONLY|O_CREAT and 0666
    BPatch_constExpr fileFlagsExpr(O_WRONLY|O_CREAT);
    BPatch_constExpr fileModeExpr(0666);

    // (5) Push 3 & 4 onto the list from step 2, push first to last parameter.
    openArgs.push_back(&fileNameExpr);
    openArgs.push_back(&fileFlagsExpr);
    openArgs.push_back(&fileModeExpr);

    // (6) create a procedure call using function found at 1 and
    // parameters from step 5.
    BPatch_funcCallExpr openCall(*openFuncs[0], openArgs);

    // (7) The oneTimeCode returns whatever the return result from
    // the BPatch_snippet is. In this case, the return result of
    // open -> the file descriptor.
    void *openFD = app->oneTimeCode( openCall );

    // oneTimeCode returns a void *, and we want an int file handle
    return (int) (long) openFD;
}

// We have used a oneTimeCode to open the file descriptor. However,
// this returns the file descriptor to the mutator - the mutatee has
// no idea what the descriptor is. We need to allocate a variable in
// the mutatee to hold this value for future use and copy the
// (mutator-side) value into the mutatee variable.

// Note: there are alternatives to this technique. We could have
// allocated the variable before the oneTimeCode and augmented the
// snippet to do the assignment. We could also write the file
// descriptor as a constant into any inserted instrumentation.

BPatch_variableExpr *writeFileDescIntoMutatee(BPatch_process *app,
                                              BPatch_image *appImage,
                                              int fileDescriptor) {
    // (1) Allocate a variable in the mutatee of size (and type) int
    BPatch_variableExpr *fdVar = app->malloc(*appImage->findType("int"));
    if (fdVar == NULL) return NULL;

    // (2) Write the value into the variable
    // Like memcpy, writeValue takes a pointer
    // The third parameter is for functionality called "saveTheWorld",
    // which we don't worry about here (and so is false)
```

```
    bool ret = fdVar->writeValue((void *) &fileDescriptor, sizeof(int),
                                                          false);
    if (ret == false) return NULL;

    return fdVar;
}

// We now have an open file descriptor in the mutatee. We want to
// instrument write to intercept and copy the output. That happens
// here.

bool interceptAndCloneWrite(BPatch_process *app,
                            BPatch_image *appImage,
                            BPatch_variableExpr *fdVar) {
    // (1) Locate the write call
    std::vector<BPatch_function *> writeFuncs;

    appImage->findFunction("write",
                           writeFuncs);
    if(writeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for write()\n");
        return false;
    }

    // (2) Build the call to (our) write. Arguments are:
    //   ours: fdVar (file descriptor)
    //   parameter: buffer
    //   parameter: buffer size

    // Declare a vector to hold these.
    std::vector<BPatch_snippet *> writeArgs;
    // Push on the file descriptor
    writeArgs.push_back(fdVar);
    // Well, we need the buffer... but that's a parameter to the
    // function we're implementing. That's not a problem - we can grab
    // it out with a BPatch_paramExpr.
    BPatch_paramExpr buffer(1); // Second (0, 1, 2) argument
    BPatch_paramExpr bufferSize(2);
    writeArgs.push_back(&buffer);
    writeArgs.push_back(&bufferSize);

    // And build the write call
    BPatch_funcCallExpr writeCall(*writeFuncs[0], writeArgs);

    // (3) Identify the BPatch_point for the entry of write. We're
    // instrumenting the function with itself; normally the findPoint
    // call would operate off a different function than the snippet.

    std::vector<BPatch_point *> *points;
    points = writeFuncs[0]->findPoint(BPatch_entry);
    if ((*points).size() == 0) {
        return false;
    }
```

```
    // (4) Insert the snippet at the start of write

    return app->insertSnippet(writeCall, *points);

    // Note: we have just instrumented write() with a call to
    // write(). This would ordinarily be a _bad thing_, as there is
    // nothing to stop infinite recursion - write -> instrumentation
    // -> write -> instrumentation....
    // However, Dyninst uses a feature called a "tramp guard" to
    // prevent this, and it's on by default.
}

// This function is called as an exit callback (that is, called
// immediately before the process exits when we can still affect it)
// and thus must match the exit callback signature:
//
// typedef void (*BPatchExitCallback) (BPatch_thread *, BPatch_exitType)
//
// Note that the callback gives us a thread, and we want a process - but
// each thread has an up pointer.

void closeFile(BPatch_thread *thread, BPatch_exitType) {
    fprintf(stderr, "Exit callback called for process...\n");

    // (1) Get the BPatch_process and BPatch_images
    BPatch_process *app = thread->getProcess();
    BPatch_image *appImage = app->getImage();

    // The code to be generated is:
    // close(fd);

    // (2) Find close
    std::vector<BPatch_function *> closeFuncs;
    appImage->findFunction("close", closeFuncs);
    if (closeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for close()\n");
        return;
    }

    // (3) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> closeArgs;

    // (4) Add the fd snippet - fdVar is global since we can't
    // get it via the callback
    closeArgs.push_back(fdVar);

    // (5) create a procedure call using function found at 1 and
    // parameters from step 3.
    BPatch_funcCallExpr closeCall(*closeFuncs[0], closeArgs);

    // (6) Use a oneTimeCode to close the file
    app->oneTimeCode( closeCall );

    // (7) Tell the app to continue to finish it off.
```

```
    app->continueExecution();

    return;
}

BPatch bpatch;

// In main we perform the following operations.
// 1) Attach to the process and get BPatch_process and BPatch_image
//    handles
// 2) Open a file descriptor
// 3) Instrument write
// 4) Continue the process and wait for it to terminate

int main(int argc, char *argv[]) {
    int pid;
    if (argc != 3) {
        usage();
        exit(1);
    }
    pid = atoi(argv[1]);

    // Attach to the program - we can attach with just a pid; the
    // program name is no longer necessary
    fprintf(stderr, "Attaching to process %d...\n", pid);
    BPatch_process *app = bpatch.processAttach(NULL, pid);

    if (!app) return -1;

    // Read the program's image and get an associated image object
    BPatch_image *appImage = app->getImage();
    std::vector<BPatch_function*> writeFuncs;

    fprintf(stderr, "Opening file %s for write...\n", argv[2]);
    int fileDescriptor = openFileForWrite(app, appImage, argv[2]);

    if (fileDescriptor == -1) {
        fprintf(stderr, "ERROR: opening file %s for write failed\n",
                argv[2]);
        exit(1);
    }

    fprintf(stderr, "Writing returned file descriptor %d into"
                    "mutatee...\n", fileDescriptor);

    // This was defined globally as the exit callback needs it.
    fdVar = writeFileDescIntoMutatee(app, appImage, fileDescriptor);
    if (fdVar == NULL) {
        fprintf(stderr, "ERROR: failed to write mutatee-side variable\n");
        exit(1);
    }

    fprintf(stderr, "Instrumenting write...\n");
    bool ret = interceptAndCloneWrite(app, appImage, fdVar);
```

```
    if (!ret) {
        fprintf(stderr, "ERROR: failed to instrument mutatee\n");
        exit(1);
    }

    fprintf(stderr, "Adding exit callback...\n");
    bpatch.registerExitCallback(closeFile);

    // Continue the execution...
    fprintf(stderr, "Continuing execution and waiting for termination\n");
    app->continueExecution();

    while (!app->isTerminated())
        bpatch.waitForStatusChange();

    printf("Done.\n");

    return 0;
}
```

B Running the Test Cases

C Common pitfalls

This appendix is designed to point out some common pitfalls that users have reported when using the Dyninst system. Many of these are either due to limitations in the current implementations, or reflect design decisions that may not produce the expected behavior from the system.

Attach followed by detach
If a mutator attaches to a mutatee, and immediately exits, the current behavior is that the mutatee is left suspended. To make sure the application continues, call detach with the appro-priate flags.

Attaching to a program that has already been modified by Dyninst
If a mutator attaches to a program that has already been modified by a previous mutator, a warning message will be issued. We are working to fix this problem, but the correct seman-tics are still being specified. Currently, a message is printed to indicate that this has been at-tempted, and the attach will fail.

Dyninst is event-driven
Dyninst must sometimes handle events that take place in the mutatee, for instance when a new shared library is loaded, or when the mutatee executes a fork or exec. Dyninst handles events when it checks the status of the mutatee, so to allow this the mutator should periodi-cally call one of the functions `BPatch::pollForStatusChange`, `BPatch::waitForStatusChange`, `BPatch_thread::isStopped`, or `BPatch_thread::isTerminated`.

Missing or out-of-date DbgHelp DLL (Windows) Dyninst requires an up-to-date DbgHelp library on Windows. See the section on Win-dows-specific architectural issues for details.

Portland Compiler Group – missing debug symbols
The Portland Group compiler (pgcc) on Linux produces debug symbols that are not read correctly by Dyninst. The binaries produced by the compiler do not contain the source file information necessary for Dyninst to assign the debug symbols to the correct module.

D References

[1] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, p. 317–329, Nov. 2000.