Dyninst Programmer's Guide

May 22, 2020

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

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

A.3 INSTRUMENTING MEMORY ACCESS

A.4 RETEE

B Running the Test Cases

C Common pitfalss

Index of terms

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.