

# Paradyn Parallel Performance Tools

## Dyninst Programmer's Guide

Release ~~8~~7.0  
~~November~~March 201~~2~~4.

Formatted: Font: Not Bold

Computer Science Department  
University of Wisconsin-Madison  
Madison, WI 537~~0~~644

Computer Science Department  
University of Maryland  
College Park, MD 20742  
Email: bugs@dyninst.org  
Web: www.dyninst.org

*Dyn  
inst*

| dyninstAPI

~~11/19/2012~~8/27/20125/11/20113/11/2011

1.	Introduction.....	6
2.	Abstractions.....	7
3.	Examples.....	9
3.1	INSTRUMENTING A FUNCTION.....	9
3.2	BINARY ANALYSIS.....	11
3.3	INSTRUMENTING MEMORY ACCESS.....	12
4.	Interface.....	14
4.1	CLASS BPATCH.....	14
4.2	CALLBACKS.....	19
4.2.1	Errors.....	20
4.2.2	Signals.....	20
4.2.3	Stopped Threads.....	21
4.2.4	Asynchronous Callbacks.....	21
4.2.5	Exec.....	22
4.2.6	Fork.....	22
4.2.7	Exit.....	23
4.2.8	Dynamic libraries.....	23
4.2.9	One time code.....	24
4.2.10	Dynamic calls.....	24
	User-triggered callbacks.....	24
4.2.11	24	
4.3	CLASS BPATCH ADDRESSSPACE.....	25
4.4	CLASS BPATCH PROCESS.....	31
4.5	CLASS BPATCH THREAD.....	33
4.6	CLASS BPATCH BINARYEDIT.....	35
4.7	CLASS BPATCH SOURCEOBJ.....	36
4.8	CLASS BPATCH FUNCTION.....	37
4.9	CLASS BPATCH POINT.....	43
4.10	CLASS BPATCH IMAGE.....	44
4.11	CLASS BPATCH MODULE.....	48
4.12	CLASS BPATCH SNIPPET.....	52
4.13	CLASS BPATCH TYPE.....	57
4.14	CLASS BPATCH VARIABLEEXPR.....	58
4.15	CLASS BPATCH FLOWGRAPH.....	59
4.16	CLASS BPATCH EDGE.....	61
4.17	CLASS BPATCH LOOPTREENODE.....	61
4.18	CLASS BPATCH BASICBLOCK.....	63
4.19	CLASS BPATCH BASICBLOCKLOOP.....	65
4.20	CLASS BPATCH INSTRUCTION.....	67
4.21	CLASS BPATCH REGISTER.....	69
4.22	CLASS BPATCH SOURCEBLOCK.....	69
4.23	CLASS BPATCH CBLOCK.....	69
4.24	CLASS BPATCH FRAME.....	70
4.25	CONTAINER CLASSES.....	70
4.25.1	Class std::vector.....	70
4.25.2	Class BPatch Set.....	71
4.26	MEMORY ACCESS CLASSES.....	72
4.26.1	Class BPatch memoryAccess.....	73
4.26.2	Class BPatch addrSpec NP.....	73
4.26.3	Class BPatch countSpec NP.....	74
4.27	TYPE SYSTEM.....	74

5.	Using DyninstAPI with the component libraries.....	79
6.	Using the API.....	79
6.1	OVERVIEW OF MAJOR STEPS.....	79
6.2	BUILDING AND INSTALLING DYNINSTAPI.....	80
6.2.1	Building on UNIX.....	80
6.2.2	Building on Windows.....	81
6.3	CREATING A MUTATOR PROGRAM.....	82
6.4	SETTING UP THE APPLICATION PROGRAM (MUTATEE).....	84
6.5	RUNNING THE MUTATOR.....	85
6.6	OPTIMIZING DYNINST PERFORMANCE.....	85
6.6.1	Optimizing Mutator Performance.....	85
6.6.2	Optimizing Mutatee Performance.....	86
	Appendix A - Complete Examples.....	89
	Appendix B - Running the Test Cases.....	97
	Appendix C - Common pitfalls.....	101
	References.....	114
1.	Introduction.....	4
2.	Abstractions.....	5
3.	Examples.....	7
3.1	INSTRUMENTING A FUNCTION.....	7
3.2	BINARY ANALYSIS.....	9
3.3	INSTRUMENTING MEMORY ACCESS.....	10
4.	Interface.....	11
4.1	CLASS BPATCH.....	11
4.1.1	Callbacks.....	16
4.2	CLASS BPATCH ADDRESSSPACE.....	20
4.3	CLASS BPATCH PROCESS.....	25
4.4	CLASS BPATCH THREAD.....	27
4.5	CLASS BPATCH BINARYEDIT.....	29
4.6	CLASS BPATCH SOURCEOBJ.....	30
4.7	CLASS BPATCH FUNCTION.....	31
4.8	CLASS BPATCH POINT.....	34
4.9	CLASS BPATCH IMAGE.....	35
4.10	CLASS BPATCH MODULE.....	38
4.11	CLASS BPATCH SNIPPET.....	41
4.12	CLASS BPATCH TYPE.....	46
4.13	CLASS BPATCH VARIABLEEXPR.....	47
4.14	CLASS BPATCH FLOWGRAPH.....	48
4.15	CLASS BPATCH EDGE.....	50
4.16	CLASS BPATCH LOOPFREENODE.....	50
4.17	CLASS BPATCH BASICBLOCK.....	52
4.18	CLASS BPATCH BASICBLOCKLOOP.....	54
4.19	CLASS BPATCH INSTRUCTION.....	55
4.20	CLASS BPATCH REGISTER.....	55

4.21	CLASS BPATCH_SOURCEBLOCK.....	56
4.22	CLASS BPATCH_CBLOCK.....	56
4.23	CLASS BPATCH_FRAME.....	56
4.24	CONTAINER CLASSES.....	57
	4.24.1 Class BPatch_Vector.....	57
	4.24.2 Class BPatch_Set.....	57
4.25	MEMORY ACCESS CLASSES.....	59
	4.25.1 Class BPatch_memoryAccess.....	59
	4.25.2 Class BPatch_addrSpec_NP.....	60
	4.25.3 Class BPatch_countSpec_NP.....	60
4.26	TYPE SYSTEM.....	60
4.27	HYBRID CODE ANALYSIS.....	61
5.	Using the API.....	65
	5.1 OVERVIEW OF MAJOR STEPS.....	65
	5.2 INSTALLING DYNINSTAPI.....	65
	5.3 CREATING A MUTATOR PROGRAM.....	68
	5.4 SETTING UP THE APPLICATION PROGRAM (MUTATEE).....	69
	5.5 RUNNING THE MUTATOR.....	70
	5.6 OPTIMIZING DYNINST PERFORMANCE.....	70
	5.6.1 Optimizing Mutator Performance.....	70
	5.6.2 Optimizing Mutatee Performance.....	71
	Appendix A—Complete Examples.....	74
	Appendix B—Running the Test Cases.....	82
	Appendix C—Common pitfalls.....	86
	References.....	91
1.	Introduction.....	4
2.	Abstractions.....	5
3.	Examples.....	7
	3.1 INSTRUMENTING A FUNCTION.....	7
	3.2 BINARY ANALYSIS.....	9
	3.3 INSTRUMENTING MEMORY ACCESS.....	10
4.	Interface.....	11
	4.1 CLASS BPATCH.....	11
	4.1.1 Callbacks.....	16
	4.2 CLASS BPATCH_ADDRESSSPACE.....	20
	4.3 CLASS BPATCH_PROCESS.....	25
	4.4 CLASS BPATCH_THREAD.....	27
	4.5 CLASS BPATCH_BINARYEDIT.....	29
	4.6 CLASS BPATCH_SOURCEOBJ.....	30

4.7	CLASS BPATCH_FUNCTION	31
4.8	CLASS BPATCH_POINT	36
4.9	CLASS BPATCH_IMAGE	37
4.10	CLASS BPATCH_MODULE	41
4.11	CLASS BPATCH_SNIPPET	44
4.12	CLASS BPATCH_TYPE	49
4.13	CLASS BPATCH_VARIABLEEXPR	50
4.14	CLASS BPATCH_FLOWGRAPH	51
4.15	CLASS BPATCH_EDGE	52
4.16	CLASS BPATCH_LOOPTREENODE	53
4.17	CLASS BPATCH_BASICBLOCK	54
4.18	CLASS BPATCH_BASICBLOCKLOOP	56
4.19	CLASS BPATCH_INSTRUCTION	57
4.20	CLASS BPATCH_REGISTER	58
4.21	CLASS BPATCH_SOURCEBLOCK	58
4.22	CLASS BPATCH_CBLOCK	58
4.23	CLASS BPATCH_FRAME	59
4.24	CLASS BPATCH_DEPENDENCEGRAPHNODE	60
4.25	CLASS BPATCH_DEPENDENCEGRAPHEDGE	61
4.26	CONTAINER CLASSES	62
	4.26.1 Class BPatch_Vector	62
	4.26.2 Class BPatch_Set	62
4.27	MEMORY ACCESS CLASSES	64
	4.27.1 Class BPatch_memoryAccess	64
	4.27.2 Class BPatch_addrSpec_NP	65
	4.27.3 Class BPatch_countSpec_NP	65
4.28	TYPE SYSTEM	65
4.29	HYBRID CODE ANALYSIS	66
5.	Using the API	69
	5.1 OVERVIEW OF MAJOR STEPS	69
	5.2 CREATING A MUTATOR PROGRAM	69
	5.3 SETTING UP THE APPLICATION PROGRAM (MUTATEE)	70
	5.4 RUNNING THE MUTATOR	71
	5.5 OPTIMIZING DYNINST PERFORMANCE	71
	5.5.1 Optimizing Mutator Performance	72
	5.5.2 Optimizing Mutatee Performance	73
	Appendix A Complete Examples	76
	Appendix B Running the Test Cases	81
	Appendix C Common pitfalls	85
	Appendix D Building Dyninst	87
	References	92

## 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 machine 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<sup>1</sup>.

---

<sup>1</sup> 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 the first instruction in the procedure 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* represents either a process (for dynamic instrumentation) or a disk executable (for static instrumentation). Dynamic instrumentation is the *address space* that represents a binary and any dynamic libraries loaded with the process. For static instrumentation, the *address space* represents an executable file on disk and the set of dynamic library files that the executable depends on. 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 *mutatee* contains debug information, DyninstAPI will also provide abstractions about variable and function types, *local variables*, *function parameters*, and *source code line information*. The collection of *functions* and *variables* in a *mutatee* 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 4.274.264.28 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

Formatted: Font: Italic

*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 provides an example of a complete Dyninst program.

#### 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 will assume that the mutator program has declared a global variable called `bpatch` of class `BPatch`.

All instrumentation will be done with a `BPatch_addressSpace` object, which allows us to write code that will work for both static and dynamic instrumentation. During initialization we will use either `BPatch_process` to create or attach to 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);
```

This creates a new instance of the `BPatch_process` class that refers to the existing process. It had no effect on the state of the process (i.e., running or stopped). If the process has not been started, the mutator specifies the pathname and argument list of a program it seeks to execute:

```
BPatch_process *appProc = bpatch.processCreate(pathname, argv);
```

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

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

The above statements will create either a `BPatch_process` object or `BPatch_binaryEdit` object, depending on whether Dyninst is doing static or dynamic 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)
```

Formatted: Code

Similarly, all instrumentation commands can be performed on a BPatch\_addressSpace object, allowing similar code to be used between dynamic instrumentation and binary rewriting:

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

Formatted: Code

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 wanted to instrument the entry point of InterestingProcedure it should get a BPatch\_function from the application's BPatch\_image, and get the entry BPatch\_instPoint from that function:

```
BPatch_Vectorstd::vector<BPatch_function *> functions;
BPatch_Vectorstd::vector<BPatch_point *> *points;

BPatch_image *appImage = app->getImage();
appImage->findFunction("InterestingProcedure", functions);
points = functions[0]->findPoint(BPatch_entry);
```

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 doing an insert-Snippet 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 static or dynamic 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 use binary rewriting as part of the analysis. 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::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::std::set<BPatch_basicBlock*> blocks;
fg->getAllBasicBlocks(blocks);
```

Each basic block has a list of instructions. Each instruction is represented by a Dyninst::InstructionAPI::Instruction::Ptr object.

```
std::std::set<BPatch_basicBlock*>::iterator block_iter;
for (block_iter = blocks.begin(); block_iter != blocks.end(); ++block_iter)
{
    BPatch_basicBlock *block = *block_iter;
    std::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 constructors; the result of the snippet depends on the instrumentation point where the snippet is inserted. `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 BPatch_Set<BPatch_opCode>&)`. For example, to instrument all the loads and stores in a function named `foo` with a call to `printf`, one may write:

```

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

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

// scan the function foo and create instrumentation points
std::vector<BPatch_function*> funcs;
img->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: %p.\n");
printfArgs.push_back(fmt);
BPatch_snippet *eae = new BPatch_effectiveAddressExpr();
printfArgs.push_back(eae);
std::vector<BPatch_function*> printfFuncs;
img->findFunction("printf", printfFuncs);

```

```
BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);  
  
// insert the snippet at the instrumentation points  
app->insertSnippet(printfCall, *points);
```

Formatted: Normal

## 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 `BPatch_Vector` `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.

```
BPatch_Vector 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_hybridMode 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 before 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; more details can be found in Section \[1.14.27\]\(#\).](#)

Field Code Changed

```
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 returns `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()
```

Returns `true` if debugger information parsing is enabled, and `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 ()
```

Returns whether trampoline recursion is enabled or disabled. `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()
```

Returns `true` if type-checking of snippets is enabled, and `false` otherwise.

```
bool waitForStatusChange()
```

This function waits until there is a status change to some thread that has not yet been reported 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 information 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()
```

Returns `true` if delayed parsing is enabled, and `false` otherwise.

```
void setInstrStackFrame(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()
```

Returns `true` if instrumentation will create stack frames, `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 floating 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()
```

Returns `true` if base trampolines are set to be deleted, `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()
```

Returns `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 [4.274-264-28](#), are used with arrays created using this function.

```
BPatch_type *createEnum(const char *name,
    BPatch_Vectorstd::vector<char *> &elementNames,
    BPatch_Vectorstd::vector<int> &elementIds)
BPatch_type *createEnum(const char *name,
    BPatch_Vectorstd::vector<char *> &elementNames)
```

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 `elementNames` 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 [4.274-264-28](#), 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,
    BPatch_Vectorstd::vector<char *> &fieldNames,
    BPatch_Vectorstd::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 [4.274-264-28](#), 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,
    BPatch_Vectorstd::vector<char *> &fieldNames,
    BPatch_Vectorstd::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.

## 4.1.14.2 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()`.

Formatted: Heading 2, H2

#### 4.2.1 Errors

```
enum BPatchErrorLevel { BPatchFatal, BPatchSerious,
    BPatchWarning, BPatchInfo };
```

```
typedef void (*BPatchErrorCallback)(BPatchErrorLevel severity,
    int number, const char * const *params)
```

~~This is the prototype for the error callback function. The severity field indicates how important the error is (from fatal to information/status). The number is a unique number that identifies this error message. Params are the parameters that describe the detail about an error, e.g., the process id where the error occurred. The number and meaning of params depends on the error. However, for a given error number the number of parameters returned will always be the same.~~

```
BPatchErrorCallback registerErrorCallback(BPatchErrorCallback
    func)
```

This function registers the error callback function with the `BPatch` class. The return value is the address of the previous error callback function. Dyninst users can change the error callback during program execution (e.g., one error callback before a GUI is initialized, and a different one after). The severity field indicates how important the error is (from fatal to information/status). The number is a unique number that identifies this error message. Params are the parameters that describe the detail about an error, e.g., the process id where the error occurred. The number and meaning of params depends on the error. However, for a given error number the number of parameters returned will always be the same.

#### 4.2.2 Signals

```
typedef void (*BPatchSignalHandlerCallback)(BPatch_point
    *at_point, long signum,
    BPatch_Vectorstd::vector<Dyninst::Address> *handlers)
```

~~This is the prototype for the signal handler callback function. The at\_point parameter indicates the point at which the signal/exception was raised, signum is the number of the signal/exception that was raised, and the handlers vector contains any registered handler(s) for the signal/exception. In Windows this corresponds to the stack of Structured Exception Handlers, while for Unix systems there will be at most one registered exception handler. As of April 15, 2008, this functionality is only fully implemented for the Windows platform.~~

Formatted: Heading 3,H3

Formatted: Definition Char Char Char Char  
Char Char Char Char Char Char

```
bool BPatchSignalHandlerCallback
    registerSignalHandlerCallback(BPatchSignalHandlerCallback
        funcb, BPatch_Set<long> *signal_numbers)
bool removeSignalHandlerCallback(BPatchSignalHandlerCallback cb);
```

This function registers the signal handler callback function with the `BPatch` class. The return value indicates success or failure. The `signal_numbers` set contains those signal numbers for which the callback will be invoked.

The `at_point` parameter indicates the point at which the signal/exception was raised, `signum` is the number of the signal/exception that was raised, and the `handlers` vector contains any registered handler(s) for the signal/exception. In Windows this corresponds to the stack of Structured Exception Handlers, while for Unix systems there will be at most one registered exception handler. This functionality is only fully implemented for the Windows platform.

### 4.2.3 Stopped Threads

~~As of April 15, 2008, this functionality is only fully implemented for the Windows platform.~~

```
typedef void (*BPatchStopThreadCallback) (BPatch_point *at_point,
    void *returnValue)
```

This is the prototype for the callback that is associated with the `stopThreadExpr` snippet class (see Section ~~4.134.114.11~~). Unlike the other callbacks in this section, `stopThreadExpr` callbacks are registered during the creation of the `stopThreadExpr` snippet type. Whenever a `stopThreadExpr` snippet executes in a given thread, the snippet evaluates the calculation snippet that `stopThreadExpr` takes as a parameter, stops the thread's execution and invokes this callback. The `at_point` parameter is the `BPatch_point` at which the `stopThreadExpr` snippet was inserted, and `returnValue` contains the computation made by the calculation snippet.

### 4.2.4 Asynchronous Callbacks

```
typedef void (*BPatchAsyncThreadEventCallback) (
    BPatch_process *proc, BPatch_thread *thread)
```

~~This is the prototype for most callback functions associated with events that occur in a thread, such as thread creation and destruction events. The `thread` parameter is the thread that triggered the event, and `proc` is the thread's containing process.~~

Formatted: Heading 3,H3

Formatted

Field Code Changed

Formatted

Formatted

Formatted

Formatted

Formatted: Heading 3,H3

```
bool registerThreadEventCallback(BPatch_asyncEventType type,
    BPatchAsyncThreadEventCallback cb)
bool removeThreadEventCallback(BPatch_asyncEventType type,
    BPatch AsyncThreadEventCallback cb)
```

~~This function registers a callback to occur whenever the process triggers a new thread event.~~ The type parameter can be either one of BPatch\_threadCreateEvent or BPatch\_threadDestroyEvent. Different callbacks can be registered for different values of type.

Formatted: Font: (Default) Courier New

Formatted: Definition List

### 4.2.5 Exec

```
typedef void (*BPatchExecCallback) (BPatch_thread *thr)
```

~~This is the prototype for the exec callback. The thr parameter is a thread in the process that called exec. You can use the BPatch\_thread::getProcess function to get the BPatch\_process that performed the exec operation.~~

```
BPatchThreadExecEventCallback registerExecCallback(
    BPatchExecCallback func) Not implemented on Windows.
```

Formatted: Heading 3,H3

### ~~4.2.6 Register a function to be called when a thread executes an exec system call. When the function is called, the thread performing the exec will be paused.~~

```
typedef void (*BPatchForkCallback) (BPatch_thread *parent,
    BPatch_thread *child);
```

This is the prototype for the pre-fork and post-fork callbacks. The parent parameter is the parent thread, and the child parameter is a BPatch\_thread in the newly created process. When invoked as a pre-fork callback, the child is NULL.

```
BPatchForkCallback registerPreForkCallback(
    BPatchForkCallback func) not implemented on Windows
```

Formatted: Indent: Left: 0"

Formatted

~~Register a function to be called when a BPatch\_thread forks a new process. This callback is invoked just before the fork is performed. When the callback is invoked, the thread performing the fork will be stopped.~~

Formatted: Indent: First line: 0"

```
BPatchPostForkCallback registerPostForkCallback(
    BPatchPostForkCallback func) not implemented on Windows
```

Register callbacks for pre-fork (before the child is created) and post-fork (immediately after the child is created). When a pre-fork callback is executed the child parameter will be NULL.

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

~~Register a function to be called just after the fork is performed. Both the thread performing the fork and the newly created thread will be paused when the callback is invoked. Unless a post fork callback is registered, the mutator will not be attached to any child processes. Since there is overhead associated with each tracked process, not setting the callback allows the Dyninst library to ignore any child processes. This is particularly useful for instrumenting shell processes that create many (potentially) uninteresting children.~~

#### 4.2.7 Exit

```
typedef enum BPatch_exitType { NoExit, ExitedNormally,
    ExitedViaSignal };
```

```
typedef void (*BPatchExitCallback) (BPatch_thread *proc,
    BPatch_exitType exit_type);
```

~~This is the prototype for the callback function called when a process exit occurs. The `proc` parameter is a thread in the process which exited. The `exit_type` parameter indicates how the process exited, either normally or because of a signal. The functions `BPatch_thread::getExitCode()` and `BPatch_thread::getExitSignal()` can be used to get further information about the process exit.~~

```
BPatchExitThreadEventCallback registerExitCallback(
    BPatchExitCallback func)
```

Register a function to be called when a process terminates. For a normal process exit, the callback will actually be called just before the process exits, but while its process state still exists, when the process is at the entry to the `exit()` function (except for Windows). This allows final actions to be taken on the process before it actually exits. The function `BPatch_thread::isTerminated()` will return true in this context even though the process hasn't yet actually exited. In the case of an exit due to a signal, the process will have already exited.

~~**4.2.8 Dynamic libraries.** On AIX, the reason why a process exited may not be available if the process was not a child of the Dyninst mutator; the mutator will be notified of the process exiting.~~

```
typedef void (*BPatchDynLibraryCallback) (Bpatch_thread *thr,
    Bpatch_module *mod, bool loaded);
```

~~This is the prototype for the dynamic linker callback function. The `thr` field contains the thread that loaded or unloaded a shared library. The `mod` field contains the module that was loaded or unloaded. The `load` Boolean is true if the library was loaded and false if it was unloaded.~~

Formatted: Heading 3,H3

Formatted: Heading 3,H3

```

BPatchDynLibraryThreadEventCallback
    registerDynLibraryCallback(
        BPatchDynLibraryThreadEventCallback func)

```

#### 4.2.9 One time code

Register a function to be called when an application has loaded or unloaded a dynamic library.

```

typedef void (*BPatchOneTimeCodeCallback) (Bpatch_thread *thr,
    void *userData, void *returnValue);

```

This is the prototype for the `oneTimeCode` callback function. The `thr` field contains the thread that executed the `oneTimeCode` (if thread-specific) or an unspecified thread in the process (if process-wide). The `userData` field contains the value passed to the `oneTimeCode` call. The `returnValue` field contains the return result of the `oneTimeCode` snippet.

```

BPatchOneTimeCodeCallback registerOneTimeCodeCallback(
    BPatchOneTimeCodeCallback func)

```

The `thr` field contains the thread that executed the `oneTimeCode` (if thread-specific) or an unspecified thread in the process (if process-wide). The `userData` field contains the value passed to the `oneTimeCode` call. The `returnValue` field contains the return result of the `oneTimeCode` snippet.

#### 4.2.10 Register a function to be called on completion of any `oneTimeCode`. Dynamic calls

```

typedef void (*BPatchDynamicCallSiteCallback) (
    BPatch point *at point, BPatch function *called function);
bool registerDynamicCallCallback(BPatchDynamicCallSiteCallback
    cb);
bool removeDynamicCallCallback(BPatchDynamicCallSiteCallback cb);

```

#### ———User-triggered callbacks

#### 4.2.11

```

typedef void (*BPatchUserEventCallback) (BPatch process *proc,
    void *buf, unsigned int bufsize);
bool registerUserEventCallback(BPatchUserEventCallback cb)
bool removeUserEventCallback(BPatchUserEventCallback cb)

```

Register a callback that is executed when the user sends a message from the mutatee using the `DYNINSTUserMessage` function in the runtime library.

Formatted: Heading 3,H3

Formatted: Font: Courier New

Formatted: Font: Courier New

Formatted: APIDesc

Formatted: Font: Courier New

Formatted: Font: Courier New

Formatted: Font: Courier New

Formatted: Font: Courier New

Formatted: Font: (Default) Courier New, 10 pt



#### 4.24.3 Class BPatch\_addressSpace

The **BPatch\_addressSpace** class is a superclass of the `BPatch_process` and `BPatch_binaryEdit` classes. It contains functionality that is common between the two sub classes.

```
const BPatch_image *getImage()
```

Return a handle to the executable file associated with this `BPatch_process` object.

```
bool getSourceLines(unsigned long addr, std::vector<  
BPatch_statement >std::vector< std::pair< const char *,  
unsigned int >> & lines)
```

This function returns the line information associated with the mutatee address, `addr`. The vector `lines` contain pairs of filenames and line numbers that are associated with `addr`. In many cases only one filename and line number is associated with an address, but certain compiler optimizations may lead to multiple filenames and lines at an address. This information is only available if the mutatee was compiled with debug information.

This function returns `true` if it was able to find any line information at `addr`, and `false` otherwise.

```
bool getAddressRanges( const char * fileName, unsigned int  
lineNo, std::std::vector< std::std::pair< unsigned long,  
unsigned long > > & ranges )
```

Given a filename and line number, `fileName` and `lineNo`, this function this function returns the ranges of mutatee addresses that implement the code range in the output parameter `ranges`. In many cases a source code line will only have one address range implementing it. However, compiler optimizations may transform this into multiple disjoint address ranges. This information is only available if the mutatee was compiled with debug information.

This function returns `true` if it was able to find any line information, `false` otherwise.

```
BPatch_variableExpr *malloc(int n)  
BPatch_variableExpr *malloc(const BPatch_type &type)
```

These two functions allocate memory. Memory allocation is from a heap. The heap is not necessarily the same heap used by the application. The available space in the heap may be limited depending on the implementation. The first function, `malloc(int n)`, allocates `n` bytes of memory from the heap. The second function, `malloc(const BPatch_type& t)`, allocates enough memory to hold an object of the specified type. Using the second version is strongly encouraged because it provides additional information to permit better type checking of the passed code. The returned memory is persistent and will not be released until `BPatch_process::free` is called or the application terminates.

```
BPatch_variableExpr *createVariable(Dyninst::Address addr,
    BPatch_type *type,
std::string var_name = std::string(""),
    BPatch_module *in_module = NULL)
```

This method creates a new variable at the given address `addr` in the module `in_module`. If a name is specified, Dyninst will assign `var_name` to the variable; otherwise, it will assign an internal name. The `type` parameter will become the type for the new variable.

When operating in binary rewriting mode, it is an error for the `in_module` parameter to be `NULL`; it is necessary to specify the module in which the variable will be created. Dyninst will then write the variable back out in the file specified by `in_module`.

```
bool void free(const BPatch_variableExpr &ptr)
```

Free the memory in the passed variable `ptr`. The programmer is responsible for verifying that all code that could reference this memory will not execute again (either by removing all snippets that refer to it, or by analysis of the program). Returns true if the free succeeded.

```
bool getRegisters(BPatch_Vectorstd::vector<BPatch_register>
    &regs) —implemented for POWER and AMD64
```

This function returns a vector of `BPatch_register` objects that represent registers available to snippet codes ~~the registers available to the mutatee.~~

~~Currently supports general purpose registers (GPRs) only. Only implemented on x86 64 and POWER.~~

```
BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
    BPatch_point &point,
    BPatch_callWhen when=[BPatch_callBefore| BPatch_callAfter],
    BPatch_snippetOrder order = BPatch_firstSnippet)
```

```
BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
    const BPatch_Vectorstd::vector<BPatch_point *> &points,
    BPatch_callWhen when=[BPatch_callBefore| BPatch_callAfter],
    BPatch_snippetOrder order = BPatch_firstSnippet)
```

Insert a snippet of code at the specified `point`. If a list of `points` is supplied, insert the code snippet at each point in the list. The optional `when` argument specifies when the snippet is to be called; a value of `BPatch_callBefore` indicates that the snippet should be inserted just before the specified `point` or `points` in the code, and a value of `BPatch_callAfter` indicates that it should be inserted just after them.

The `order` argument specifies where the snippet is to be inserted relative to any other snippets previously inserted at the same point. The values `BPatch_firstSnippet` and

`BPatch_lastSnippet` indicate that the snippet should be inserted before or after all snippets, respectively.

It is illegal to use `BPatch_callAfter` with a `BPatch_entry` point. Use `BPatch_callBefore` when instrumenting entry points, which inserts instrumentation before the first instruction in a subroutine. Likewise, it is illegal to use `BPatch_callBefore` with a `BPatch_exit` point. Use `BPatch_callAfter` with exit points. `BPatch_callAfter` inserts instrumentation at the last instruction in the subroutine. `insertSnippet` will return `NULL` when used with an illegal pair of points.

```
bool deleteSnippet(BPatchSnippetHandle *handle)
```

Remove the snippet associated with the passed `handle`. If the `handle` is not defined for the process, then `deleteSnippet` will return false.

```
bool void beginInsertionSet()
```

Normally, a call to `insertSnippet` immediately injects instrumentation into the mutatee. However, users may wish to insert a set of snippets as a single batch operation. This provides two benefits: First, Dyninst may insert instrumentation in a more efficient manner. Second, multiple snippets may be inserted at multiple points as a single operation, with either all snippets being inserted successfully or none. This batch insertion mode is begun with a call to `beginInsertionSet`; after this call, no snippets are actually inserted until a corresponding call to `finalizeInsertionSet`. Dyninst accumulates all calls to `insertSnippet` during batch mode internally, and the returned `BPatchSnippetHandles` are filled in when `finalizeInsertionSet` is called.

Insertion sets are unnecessary when doing static binary instrumentation. Dyninst uses an implicit insertion set around all instrumentation to a static binary.

```
bool finalizeInsertionSet(bool atomic)
```

Inserts all snippets accumulated since a call to `beginInsertionSet`. If the `atomic` parameter is `true`, then a failure to insert any snippet results in all snippets being removed; effectively, the insertion is all-or-nothing. If the `atomic` parameter is `false`, then snippets are inserted individually. This function also fills in the `BPatchSnippetHandle` structures returned by the `insertSnippet` calls comprising this insertion set. It returns `true` on success and `false` if there was an error inserting any snippets.

Insertion sets are unnecessary when doing static binary instrumentation. Dyninst uses an implicit insertion set around all instrumentation to a static binary.

```
bool removeFunctionCall(BPatch_point &point)
```

Disable the mutatee function call at the specified location. The `point` specified must be a valid call point in the image of the mutatee. The purpose of this routine is to permit tools to alter the semantics of a program by eliminating procedure calls. The mechanism to

achieve the removal is platform dependent, but might include branching over the call or replacing it with NOPs. This function only removes a function call; any parameters to the function will still be evaluated.

```
bool replaceCode(BPatch_point *point, BPatch_snippet *snippet)
```

~~Replaces the instruction identified by `point` with the provided snippet. The provided `point` may represent either a specific instruction (acquired via `BPatch_function::findPoint`), function entry, function exit, or a call site. If the point corresponds to a function entry, the first instruction in the function will be replaced. If the point corresponds to a function exit, the return instruction at that exit will be replaced. If the point corresponds to a call site, the call instruction will be replaced. If the point corresponds to any other location, replacement will fail and an error will be returned.~~

~~This call returns `true` if the replacement succeeded, or `false` otherwise. The replacement mechanism uses similar techniques as our instrumentation mechanism, and can fail in the same circumstances.~~

~~**WARNING:** This function is dangerous. Unlike instrumentation, program state is not saved and restored around the new code. The provided snippet may modify registers and memory. This call may have unexpected effects on program execution, resulting in incorrect results or program crashes.~~

Formatted: APIDesc

```
bool replaceFunction (BPatch_function &old, BPatch_function &new)
bool revertReplaceFunction (BPatch_function &old)
```

Replace all calls to user function `old` with calls to `new`. This is done by inserting instrumentation (specifically a `BPatch_funcJumpExpr`) into the beginning of function `old` such that a non-returning jump is made to function `new`. Returns `true` upon success, `false` otherwise.

```
bool replaceFunctionCall(BPatch_point &point, BPatch_function &newFunc)
```

Change the function call at the specified `point` to the function indicated by `newFunc`. The purpose of this routine is to permit runtime steering tools to change the behavior of programs by replacing a call to one procedure by a call to another. `Point` must be a function call point. If the change was successful, the return value is `true`, otherwise `false` will be returned.

**WARNING:** Care must be used when replacing functions. In particular if the compiler has performed inter-procedural register allocation between the original caller/callee pair, the replacement may not be safe since the replaced function may clobber registers

*the compiler thought the callee left untouched. Also the signatures of the both the function being replaced and the new function must be compatible.*

```
bool wrapFunction(BPatch function *old, BPatch function *new,
    Dyninst::SymtabAPI::Symbol *sym)
bool revertWrapFunction(BPatch function *old)
```

Replaces all calls to function `old` with calls to function `new`. Unlike `replaceFunction` above, the old function can still be reached via the name specified by the provided symbol `sym`. Function wrapping allows existing code to be extended by new code. Consider the following code that implements a fast memory allocator for a particular size of memory allocation, but falls back to the original memory allocator (referenced by `origMalloc`) for all others.

```
void *origMalloc(unsigned long size);

void *fastMalloc(unsigned long size) {
    if (size == 1024) {
        unsigned long ret = fastPool;
        fastPool += 1024;
        return ret;
    }
    else {
        return origMalloc(size);
    }
}
```

The symbol `sym` is provided by the user and must exist in the program; the easiest way to ensure it is created is to use an undefined function as shown above with the definition of `origMalloc`.

The following code wraps `malloc` with `fastMalloc`, while allowing functions to still access the original `malloc` function by calling `origMalloc`. It makes use of the `new` convert interface described in Section 5.

```
using namespace Dyninst;
using namespace SymtabAPI;
BPatch function *malloc = appImage->findFunction(...);
BPatch function *fastMalloc = appImage->findFunction(...);
Symtab *symtab = SymtabAPI::convert(fastMalloc->getModule());
std::vector<Symbol*> syms;
symtab->findSymbol(syms, "origMalloc",
    Symbol::ST_UNKNOWN, // Don't specify type
    mangledName, // Look for raw symbol name
    false, // Not regular expression
    false, // Don't check case
    true); // Include undefined symbols
app->wrapFunction(malloc, fastMalloc, syms[0]);
```

For a full, executable example, see Appendix A.

Formatted: Font: (Default) Courier New, 10 pt

Formatted: APIDesc

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: Courier New

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: Courier New

Formatted: Font: (Default) Courier New, 10 pt

Field Code Changed

```
bool replaceCode(BPatch point *point, BPatch snippet *snippet)
```

~~This function has been removed; users interested in replacing code should instead use the PatchAPI code modification interface described in the PatchAPI manual. For information on accessing PatchAPI abstractions from DyninstAPI abstractions, see Section 5.~~

```
BPatch module *bool loadLibrary(const char *libname, bool  
    reload=false)
```

For dynamic rewriting, this function loads a dynamically linked library into the process's address space. For static rewriting, this function adds a library as a library dependency in the rewritten file. In both cases Dyninst creates a new `BPatch_module` to represent this library.

The `libname` parameter identifies the file name of the library to be loaded, in the standard way that dynamically linked libraries are specified on the operating system on which the API is running. This function ~~returns a handle to the loaded library. returns true if the library was loaded successfully, otherwise it returns false.~~

The `reload` parameter is ignored and only remains for backwards compatibility.

```
bool isStaticExecutable()
```

This function returns `true` if the original file opened with this `BPatch_addressSpace` is a statically linked executable, and `false` otherwise.

```
void allowTraps(bool allowtraps)
```

~~This function is used to tell Dyninst whether it can fall back to use traps when doing instrumentation. Depending on certain architecture dependent characteristics, certain functions may be difficult to instrument, and Dyninst must fall back to inserting a trap to do instrumentation. This can have a serious performance impact on the mutatee.~~

~~If this function is called with `allowtraps` set to `false`, then Dyninst will not insert any instrumentation that depends on a trap. If a piece of instrumentation would depend on a trap, the `insertSnippet` will return an error instead of inserting it. If this function is called with `allowtraps` set to `true`, then Dyninst will use trap-based instrumentation if necessary.~~

~~The default value for `allowTraps` is `true`.~~

Formatted: APIDesc

Formatted: Indent: Left: 0", First line: 0"

Formatted

Formatted

Formatted: Indent: Left: 0"

Formatted

Formatted

Formatted

Formatted

Formatted

#### 4.34.4 Class BPatch\_process

The **BPatch\_process** class represents a running process, which includes one or more threads of execution and an address space.

```
bool stopExecution()
bool continueExecution()
bool terminateExecution()
```

These three functions change the running state of the process. `stopExecution` puts the process into a stopped state. Depending on the operating system, stopping one process may stop all threads associated with a process. `continueExecution` continues execution of the process. `terminateExecution` terminates execution of the process and will invoke the exit callback if one is registered. Each function returns true on success, or false for failure. Stopping or continuing a terminated thread will fail and these functions will return false.

```
bool isStopped()
int stopSignal()
bool isTerminated()
```

These three functions query the status of a process. `isStopped` returns true if the process is currently stopped. If the process is stopped (as indicated by `isStopped`), then `stopSignal` can be called to find out what signal caused the process to stop. `isTerminated` returns true if the process has exited. Any of these functions may be called multiple times, and calling them will not affect the state of the process.

```
BPatch_variableExpr *getInheritedVariable(const
    BPatch_variableExpr &parentVar)
```

Retrieve a new handle to an existing variable (such as one created by `BPatch_process::malloc`) that was created in a parent process and now exists in a forked child process. When a process forks all existing `BPatch_variableExpr`s are copied to the child process, but the Dyninst handles for these objects are not valid in the child `BPatch_process`. This function is invoked on the child process' `BPatch_process`, `parentVar` is a variable from the parent process, and a handle to a variable in the child process is returned. If `parentVar` was not allocated in the parent process, then `NULL` is returned.

```
BPatchSnippetHandle *getInheritedSnippet(BPatchSnippetHandle
    &parentSnippet)
```

This function is similar to `getInheritedVariable`, but operates on `BPatchSnippetHandles`. Given a child process that was created via fork and a `BPatchSnippetHandle`, `parentSnippet`, from the parent process, this function will return a handle to par-

entSnippet that is valid in the child process. If it is determined that parentSnippet is not associated with the parent process, then NULL is returned.

~~void setMutationsActive(bool)~~

~~Enable or disable the execution of snippets for the process. This provides a way to temporarily disable all of the dynamic code patches that have been inserted without having to delete them one by one. All allocated memory will remain unchanged while the patches are disabled. When the mutations are not active, any process control functions (i.e., stopExecution and continueExecution) may still be used. Requests to insert snippets (including oneTimeCode) cannot be made while mutations are disabled.~~

~~void~~ detach(bool cont)

Detach from the process. The process must be stopped to call this function. Instrumentation and other changes to the process will remain active in the detached copy. The cont parameter is used to indicate if the process should be continued as a result of detaching.

Linux does not support detaching from a process while leaving it stopped. All processes are continued after detach on Linux.

int getpid()

Return the system id for the mutatee process. On UNIX based systems this is a PID. On Windows this is the HANDLE object for a process.

```
typedef enum BPatch_exitType { NoExit, ExitedNormally,
                               ExitedViaSignal };
```

BPatch\_exitType terminationStatus()

If the process has exited, terminationStatus will indicate whether the process exited normally or because of a signal. If the process has not exited, NoExit will be returned. On AIX, the reason why a process exited will not be available if the process was not a child of the Dyninst mutator; in this case, ExitedNormally will be returned in both normal and signal exit cases.

int getExitCode()

If the process exited in a normal way, getExitCode will return the associated exit code. On AIX, this code will not be available if the process was not a child of the Dyninst mutator.

int getExitSignal()

If the process exited because of a received signal, getExitSignal will return the associated signal number. On AIX, this code will not be available if the process was not a child of the Dyninst mutator.



```
void oneTimeCode(const BPatch_snippet &expr)
```

Cause the snippet `expr` to be executed by the mutatee immediately. If the process is multithreaded, the snippet is run on a thread chosen by Dyninst. If the user requires the snippet to be run on a particular thread, use the `BPatch_thread` version of this function instead. The process must be stopped to call this function. The behavior is synchronous; `oneTimeCode` will not return until after the snippet has been run in the application.

```
bool oneTimeCodeAsync(const BPatch_snippet &expr,
                     void *userData = NULL)
```

This function sets up a snippet to be evaluated by the process at the next available opportunity. When the snippet finishes running Dyninst will callback any function registered through `BPatch::registerOneTimeCodeCallback`, with `userData` passed as a parameter. This function return `true` on success and `false` if it could not post the `oneTimeCode`.

If the process is multithreaded, the snippet is run on a thread chosen by Dyninst. If the user requires the snippet to be run on a particular thread, use the `BPatch_thread` version of this function instead. The behavior is asynchronous; `oneTimeCodeAsync` returns before the snippet is executed.

If the process is running when `oneTimeCodeAsync` is called, `expr` will be run immediately. If the process is stopped, then `expr` will be run when the process is continued.

```
void getThreads(std::vector<BPatch_thread *> &thrds)
```

Get the list of threads in the process.

```
bool isMultithreaded()  
bool isMultithreadCapable()
```

The former returns true if the process contains multiple threads; the latter returns true if the process can create threads (e.g., it contains a threading library) even if it has not yet.

#### 4.44.5 Class BPatch\_thread

The **BPatch\_thread** class represents and controls a thread of execution that is running in a process.

```
voidvoid getCallStack(BPatch_Vectorstd::vector<BPatch_frame>&  
stack)
```

This function fills the given vector with current information about the call stack of the thread. Each stack frame is represented by a `BPatch_frame` (see section [4.244.234.23](#) for information about this class).

Formatted: API

Field Code Changed

Formatted

Formatted

dynthread\_t~~long~~ getTid()

This function returns a platform-specific identifier for this thread. This is the identifier that is used by the threading library. For example, on pthread applications this function will return the thread's `pthread_t` value.

Dyninst::LWP~~long~~ getLWP()

This function returns a platform-specific identifier that the operating system uses to identify this thread. For example, on UNIX platforms this returns the LWP id. On Windows this returns a HANDLE object for the thread.

unsigned~~long~~ getBPatchID()

This function returns a Dyninst-specific identifier for this thread. These ID's apply only to running threads, the BPatch ID of an already terminated thread may be repeated in a new thread.

BPatch\_function \*getInitialFunction()

Return the function that was used by the application to start this thread. For example, on pthread applications this will return the initial function that was passed to `pthread_create`.

unsigned long getStackTopAddr()

Returns the base address for this thread's stack.

bool isDeadOnArrival()

This function returns true if this thread terminated execution before Dyninst was able to attach to it. Since Dyninst performs new thread detection asynchronously, it is possible for a thread to be created and destroyed before Dyninst can attach to it. When this happens, a new `BPatch_thread` is created, but `isDeadOnArrival` always returns true for this thread. It is illegal to perform any thread-level operations on a dead on arrival thread.

BPatch\_process \*getProcess()

Return the `BPatch_process` that contains this thread.

void \*oneTimeCode(const BPatch\_snippet &expr, bool \*err = NULL)

Cause the snippet `expr` to be evaluated by the process immediately. This is similar to the `BPatch_process::oneTimeCode` function, except that the snippet is guaranteed to run only

on this thread. The process must be stopped to call this function. The behavior is synchronous; `oneTimeCode` will not return until after the snippet has been run in the application.

```
bool oneTimeCodeAsync(const BPatch_snippet &expr,
                     void *userData ==
                     NULL,
                     BpatchOneTimeCodeCallback cb = NULL)
```

Formatted: Indent: Left: 1.5", First line: 0.5"

This function sets up the snippet `expr` to be evaluated by this thread at the next available opportunity. When the snippet finishes running, Dyninst will callback any function registered through `BPatch::registerOneTimeCodeCallback`, with `userData` passed as a parameter. This function returns `true` if `expr` was posted and `false` otherwise.

This is similar to the `BPatch_process::oneTimeCodeAsync` function, except that the snippet is guaranteed to run only on this thread. The process must be stopped to call this function. The behavior is asynchronous; `oneTimeCodeAsync` returns before the snippet is executed.

#### 4.54.6 Class `BPatch_binaryEdit`

The `BPatch_binaryEdit` class represents a set of executable files and library files for binary rewriting. `BPatch_binaryEdit` inherits from the `BPatch_addressSpace` class, where most functionality for binary rewriting is found.

```
bool writeFile(const char *outFile)
```

Rewrite a `BPatch_binaryEdit` to disk. The original file opened with this `BPatch_binaryEdit` is written to the current working directory with the name `outFile`. If any dependent libraries were also opened and have instrumentation or other modifications, then those libraries will be written to disk in the current working directory under their original names.

A rewritten dependency library should only be used with the original file that was opened for rewriting. For example, if the file `a.out` and its dependent library `libfoo.so` were opened for rewriting, and both had instrumentation inserted, then the rewritten `libfoo.so` should not be used without the rewritten `a.out`. To build a rewritten `libfoo.so` that can load into any process, `libfoo.so` must be the original file opened by `BPatch::openBinary`.

This function returns `true` if it successfully wrote a file, and `false` otherwise.

#### 4.64.7 Class BPatch\_sourceObj

The BPatch\_sourceObj class is the C++ superclass for the BPatch\_function, BPatch\_module, and BPatch\_image classes. It provides a set of common methods for all three classes. In addition, it can be used to build a “generic” source navigator using the `getObjParent` and `getSourceObj` methods to get parents and children of a given level (i.e. the parent of a module is an image, and the children will be the functions).

```
enum BPatchErrorLevel { BPatchFatal, BPatchSerious,
    BPatchWarning, BPatchInfo };
```

```
enum BPatch_sourceType {
    BPatch_sourceUnknown,
    BPatch_sourceProgram,
    BPatch_sourceModule,
    BPatch_sourceFunction,
    BPatch_sourceOuterLoop,
    BPatch_sourceLoop,
    BPatch_sourceStatement };
```

```
BPatch_sourceType getSrcType()
```

Returns the type of the current source object.

```
void getSourceObj(BPatch_Vectorstd::vector<BPatch_sourceObj *>
    &objs)
```

Returns the child source objects of the current source object. For example, when called on a BPatch\_sourceProgram object this will return objects of type BPatch\_sourceFunction. When called on a BPatch\_sourceFunction object it may return BPatch\_sourceOuterLoop and BPatch\_sourceStatement objects.

```
BPatch_sourceObj *getObjParent()
```

Return the parent source object of the current source object. The parent of a BPatch\_image is NULL.

```
typedef enum BPatch_language {
    BPatch_c,
    BPatch_cPlusPlus,
    BPatch_fortran,
    BPatch_fortran77,
    BPatch_fortran90,
    BPatch_f90_demangled_stabstr,
    BPatch_fortran95,
    BPatch_assembly,
    BPatch_mixed,
    BPatch_hpf,
    BPatch_java,
    BPatch_unknownLanguage
} BPatch_language;
```

```
BPatch_language getLanguage()
```

Return the source language of the current `BPatch_sourceObject`. For programs that are written in more than one language, `BPatch_mixed` will be returned. If there is insufficient information to determine the language, `BPatch_unknownLanguage` will be returned.

#### 4.74.8 Class `BPatch_function`

An object of this class represents a function in the application. A `BPatch_image` object (see description below) can be used to retrieve a `BPatch_function` object representing a given function.

```
std::string getName();
std::string getDemangledName();
std::string getMangledName();
std::string getTypedName();
void getNames(std::vector<std::string> &names);
void getDemangledNames(std::vector<std::string> &names);
void getMangledNames(std::vector<std::string> &names);
void getTypedNames(std::vector<std::string> &names);
```

Return name(s) of the function. The `getName` functions return the primary name; this is typically the first symbol we encounter while parsing the program; `getName` is an alias for `getDemangledName`. The `getNames` functions return all known names for the function, including any names specified by weak symbols.

```
char *getName(char *buffer, int len)
```

Place the name of the function in `buffer`, up to `len` characters. It returns the value of the `buffer` parameter.

```
char *getMangledName(char *buffer, int len)
```

Formatted: API

Formatted: APIDesc

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt

~~Place the mangled (internal symbol) name of the function in `buffer`, up to `len` characters. It returns the value of the `buffer` parameter.~~

~~`char *getTypedName(char *buffer, int len)`~~

~~Place the full function prototype (from debug information) of the function in `buffer`, up to `len` characters. It returns the value of the `buffer` parameter.~~

~~`bool getNames (BPatch_vector<const char*> &names)`~~

~~Add all known names of the function to the vector `names`, including names generated by weak symbols. It returns `true` if one or more names were added. And, `false` otherwise. The names reside in memory managed by `Dyninst`.~~

~~`bool getMangledNames (BPatch_vector<const char*> &names)`~~

~~As above, but returns all known mangled (internal symbol) names.~~

~~`bool getTypedNames (BPatch_vector<const char*> &names)`~~

~~As above, but returns all known function prototypes.~~

~~`bool getAddressRange (Dyninst::Address &start,  
Dyninst::Address &end)`~~

~~Returns the bounds of the function; for non-contiguous functions, this is the lowest and highest address of code that the function includes.~~

~~`BPatch_Vector<std::vector<BPatch_localVar*>*> *getParams()`~~

~~Return a vector of `BPatch_localVar` snippets that refer to the parameters of this function. The position in the vector corresponds to the position in the parameter list (starting from zero). The returned local variables can be used to check the types of functions, and can be used in snippet expressions.~~

`BPatch_type *getReturnType()`

Return the type of the return value for this function.

~~`BPatch_variableExpr *getFunctionRef()`~~

~~For platforms with complex function pointers (e.g., 64-bit PPC) this constructs and returns the appropriate descriptor.~~

Formatted: APIDesc

Formatted: API

~~BPatch\_Vector~~std::vector<BPatch\_localVar \*> \*getVars()

Returns a vector of BPatch\_localVar objects that contain the local variables in this function. These BPatch\_localVars can be used as parts of snippets in instrumentation. This function requires debug information to be present in the mutatee. If Dyninst was unable to find any local variables, this function will return an empty vector. It is up to the user to free the vector returned by this function.

bool isInstrumentable()

Return true if the function can be instrumented, and false if it cannot. Various conditions can cause a function to be uninstrumentable. For example, there exists a platform-specific minimum function size beyond which a function cannot be instrumented.

bool isSharedLib()

This function returns true if the function is defined in a shared library.

~~const char \*~~libraryName()

~~Return the name of the library that contains this function. If the function is not defined in a library, a NULL will be returned.~~

~~BPatch\_module~~ \*getModule()

Return the module that contains this function. Depending on whether the program was compiled for debugging or the symbol table stripped, this information may not be available. This function returns NULL if module information was not found.

char \*getModuleName(char \*name, int maxLen)

Copies the name of the module that contains this function into the buffer pointed to by name. Copies at most maxLen characters and returns a pointer to name.

```
enum BPatch_procedureLocation {
    BPatch_entry,
    BPatch_exit,
    BPatch_subroutine,
    BPatch_locInstruction,
    BPatch_locBasicBlockEntry,
    BPatch_locLoopEntry,
    BPatch_locLoopExit,
    BPatch_locLoopStartIter,
    BPatch_locLoopStartExit,
    BPatch_allLocations }

```

const ~~BPatch\_Vector~~std::vector<BPatch\_point \*> \*findPoint(const BPatch\_procedureLocation loc)

Return the BPatch\_point or list of BPatch\_points associated with the procedure. It is used to select which type of points associated with the procedure will be returned. BPatch\_entry and BPatch\_exit request respectively the entry and exit points of the

Formatted: API

subroutine. `BPatch_subroutine` returns the list of points where the procedure calls other procedures. If the lookup fails to locate any points of the requested type, `NULL` is returned.

```
enum BPatch_opCode { BPatch_opLoad, BPatch_opStore,
    BPatch_opPrefetch }
BPatch_Vector<std::vector<BPatch_point *> *> *findPoint(const
    BPatch_Set<BPatch_opCode>& ops)
```

Return the vector of `BPatch_points` corresponding to the set of machine instruction types described by the argument. This version is used primarily for memory access instrumentation. The `BPatch_opCode` is an enumeration of instruction types that may be requested: `BPatch_opLoad`, `BPatch_opStore`, and `BPatch_opPrefetch`. Any combination of these may be requested by passing an appropriate argument set containing the desired types. The instrumentation points created by this function have additional memory access information attached to them. This allows such points to be used for memory access specific snippets (e.g. effective address). The memory access information attached is described under Memory Access classes in section [4.26.14.25-14.27.1](#).

Field Code Changed

```
BPatch_localVar *findLocalVar(const char *name)
```

Search the function's local variable collection for `name`. This returns a pointer to the local variable if a match is found. This function returns `NULL` if it fails to find any variables.

```
BPatch_Vector<std::vector<BPatch_variableExpr *> *>
    *findVariable(const char * name)
bool findVariable(const char *name,
    std::vector<BPatch_variableExpr> &vars)
```

Return a set of variables matching `name` at the scope of this function. If no variables match in the local scope, then the global scope will be searched for matches. This function returns `NULL` if it fails to find any variables.

```
BPatch_localVar *findLocalParam(const char *name)
```

Search the function's parameters for a given name. A `BPatch_localVar *` pointer is returned if a match is found, and `NULL` is returned otherwise.

Formatted: Font: (Default) Courier New, 10 pt

```
void *getBaseAddr()
```

Return the starting address of the function in the mutatee's address space.

```
BPatch_flowGraph *getCFG()
```

Return the control flow graph for the function, or `NULL` if this information is not available. The `BPatch_flowGraph` is described in section [4.164.144.14](#).

Field Code Changed



```
| bool findOverlapping(BPatch_Vectorstd::vector<BPatch_function *>
  &funcs)
```

Determine which functions overlap with the current function (see Section ~~22~~). Return `true` if other functions overlap the current function; the overlapping functions are added to the `funcs` vector. Return `false` if no other functions overlap the current function.

**Formatted:** Definition List

<b>DEPRECATED</b>	<del>BPatch_dependenceGraphNode* getDataDependenceGraph (BPatch_instruction* inst) implemented on x86</del>
<b>DEPRECATED</b>	<del>Returns a handle to the BPatch_dependenceGraphNode object used to navigate through the partial data dependence graph that includes the instruction inst and its predecessors, or NULL if this information is not available. A data dependence graph reflects the data dependence relations between instructions. Currently, data dependence graph does not take aliases into consideration.</del>
<b>DEPRECATED</b>	<del>BPatch_dependenceGraphNode* getControlDependenceGraph (BPatch_instruction* inst) implemented on x86</del>
<b>DEPRECATED</b>	<del>Returns a handle to the BPatch_dependenceGraphNode object used to navigate through the partial control dependence graph that includes the instruction inst and its predecessors, or NULL if this information is not available. A control dependence graph reflects the control dependence relations between instructions.</del>
<b>DEPRECATED</b>	<del>BPatch_dependenceGraphNode* getProgramDependenceGraph (BPatch_instruction* inst) implemented on x86</del>
<b>DEPRECATED</b>	<del>Returns a handle to the BPatch_dependenceGraphNode object used to navigate through the partial program dependence graph that includes the instruction inst and its predecessors, or NULL if this information is not available. A program dependence graph is merely a union of data and control dependence graphs. Currently, program dependence graph does not take aliases into consideration.</del>
<b>DEPRECATED</b>	<del>BPatch_dependenceGraphNode* getSlice(BPatch_instruction* inst) implemented on x86</del>
<b>DEPRECATED</b>	<del>Returns a handle to the BPatch_dependenceGraphNode object used to navigate through the intraprocedural slice of the function</del>

	<del>backwards from instruction <i>inst</i>. Returns NULL if this information is not available. Currently, slice of a function does not take aliases into consideration.</del>
--	--

#### 4.84.9 Class BPatch\_point

An object of this class represents a location in an application's code at which the library can insert instrumentation. A `BPatch_image` object (see section [4.104.94.9](#)) is used to retrieve a `BPatch_point` representing a desired point in the application.

```
enum BPatch_procedureLocation { BPatch_entry, BPatch_exit,
    BPatch_subroutine, BPatch_address }
BPatch_procedureLocation getPointType()
```

Return the type of the point.

```
BPatch_function *getCalledFunction()
```

Return a `BPatch_function` representing the function that is called at the point. If the point is not a function call site or the target of the call cannot be determined, then this function returns `NULL`.

```
std::std::string getCalledFunctionName()
```

Returns the name of the function called at this point. This method is similar to `getCalledFunction()->getName()`, except in cases where DyninstAPI is running in binary rewriting mode and the called function resides in a library or object file that DyninstAPI has not opened. In these cases, Dyninst is able to determine the name of the called function, but is unable to construct a `BPatch_function` object.

```
BPatch_function *getFunction()
```

Returns a `BPatch_function` representing the function in which this point is contained.

```
BPatch_basicBlockLoop *getLoop()
```

Returns the containing `BPatch_basicBlockLoop` if this point is part of loop instrumentation. Returns `NULL` otherwise.

```
void *getAddress()
```

Return the address of the first instruction at this point.

```
bool usesTrap_NP()
```

Return `true` if inserting instrumentation at this point requires using a trap. On the x86 architecture, because instructions are of variable size, the instruction at a point may be too small for Dyninst to replace it with the normal code sequence used to call instrumentation. Also, when instrumentation is placed at points other than subroutine entry, exit, or call points, traps may be used to ensure the instrumentation fits. In this case, Dyninst re-

places the instruction with a single-byte instruction that generates a trap. A trap handler then calls the appropriate instrumentation code. Since this technique is used only on some platforms, on other platforms this function always returns `false`.

```
const BPatch_memoryAccess* getMemoryAccess()
```

Returns the memory access object associated with this point. Memory access points are described in section [4.26.14.25-14.27.1](#).

```
const BPatch_Vectorstd::vector<BPatchSnippetHandle*>
    getCurrentSnippets()
const BPatch_Vectorstd::vector<BPatchSnippetHandle*>
    getCurrentSnippets(BPatch_callWhen when)
```

Return the `BPatchSnippetHandles` for the `BPatch_snippets` that are associated with the point. If argument `when` is `BPatch_callBefore`, then `BPatchSnippetHandles` for snippets installed immediately before this point will be returned. Alternatively, if `when` is `BPatch_callAfter`, then `BPatchSnippetHandles` for snippets installed immediately after this point will be returned.

```
bool getLiveRegisters(BPatch_Vectorstd::vector<BPatch_Rregister>
    &regs) implemented for POWER and AMD64
```

Fill `regs` with the registers that are live before this point (e.g., `BPatch_callBefore`). Currently returns only general purpose registers (GPRs).

```
bool isDynamic()
```

This call returns `true` if this is a dynamic call site (e.g. a call site where the function call is made via a function pointer).

```
Dyninst::InstructionAPI::Instruction::Ptr getInstructionAtPoint()
implemented for IA32, AMD64, and POWER
```

On implemented platforms, this function returns a shared pointer to an `InstructionAPI` `Instruction` object representing the first machine instruction at this point's address. On unimplemented platforms, returns a `NULL` shared pointer.

#### 4.94.10 Class `BPatch_image`

This class defines a program image (the executable associated with a process). The only way to get a handle to a `BPatch_image` is via the `BPatch_process` member function `getImage`.

```
const BPatch_point *createInstPointAtAddr (caddr_t address)
```

~~Return an instrumentation point at the specified address. This function is designed to permit users who wish to insert instrumentation at an arbitrary place in the code segment. Instruction addresses can be found using the `BPatch_instruction` object (see section [4.194.19](#)). On x86 platforms, users should take care to ensure that the requested point is~~

~~not in the middle of a multi-byte instruction. This function has been removed because it is not safe to use. Instead, use `findPoints`:~~

```
bool findPoints(Dyninst::Address addr,  
std::vector<BPatch point *> &points);
```

~~Returns a vector of `BPatch` points that correspond with the provided address, one per function that includes an instruction at that address. There will be one element if there is not overlapping code.~~

Formatted: Font: (Default) Courier New, 10 pt

Formatted: API

Formatted: Font: (Default) Courier New, 10 pt

Formatted: APIDesc, Adjust space between Latin and Asian text, Adjust space between Asian text and numbers

Formatted: APIDesc

```
BPatch_Vectorstd::vector<BPatch_variableExpr *>  
*getGlobalVariables()
```

Return a vector of global variables that are defined in this image.

```
BPatch_process *getProcess()
```

Returns the `BPatch_process` associated with this image.

```
char *getProgramFileName(char *name, unsigned int len)
```

Fills provided buffer `name` with the program's file name up to `len` characters. The file-name may include path information.

```
bool getSourceObj(BPatch_Vectorstd::vector<BPatch_sourceObj *>  
&sources)
```

Fill `sources` with the source objects (see section [4.64-54.5](#)) that belong to this image. If there are no source objects, the function returns `false`. Otherwise, it returns `true`.

```
const BPatch_Vectorstd::vector<BPatch_function *> *getProcedures(  
bool incUninstrumentable = false)
```

Return a vector of the functions in the image. If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

```
void getObjects(std::vector<BPatch object *> &objs)
```

Fill in a vector of objects in the image.

Field Code Changed

Formatted: APIDesc

```
const BPatch_Vectorstd::vector<BPatch_module *> *getModules()
```

Return a vector of the modules in the image.

```
bool getVariables(BPatch_Vectorstd::vector<BPatch_variableExpr *>
                &vars)
```

Fills vars with the global variables defined in this image. If there are no variable, the function returns false. Otherwise, it returns true.

```
BPatch_Vectorstd::vector<BPatch_function*> *findFunction(
    const char *name,
    BPatch_Vectorstd::vector<BPatch_function*> &funcs,
    bool showError = true,
    bool regex_case_sensitive = true,
    bool incUninstrumentable = false,
    bool dont_use_regex = false)
```

Return a vector of BPatch\_functions corresponding to name, or NULL if the function does not exist. If name contains a POSIX-extended regular expression, and dont\_use\_regex is false, a regular expression search will be performed on function names and matching BPatch\_functions returned. If showError is true, then Dyninst will report an error via the BPatch::registerErrorCallback if no function is found.

If the incUninstrumentable flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[NOTE: If name is not found to match any demangled function names in the module, the search is repeated as if name is a mangled function name. If this second search succeeds, functions with mangled names matching name are returned instead.]

```
BPatch_Vectorstd::vector<BPatch_function*> *findFunction(
    BPatch_Vectorstd::vector<BPatch_function*> &funcs,
    BPatchFunctionNameSieve bpsieve,
    void *sieve_data = NULL,
    int showError = 0,
    bool incUninstrumentable = false)
```

Return a vector of BPatch\_functions according to the generalized user-specified filter function bpsieve. This permits users to easily build sets of functions according to their own specific criteria. Internally, for each BPatch\_function f in the image, this method makes a call to bpsieve(f.getName(), sieve\_data). The user-specified function bpsieve is responsible for taking the name argument and determining if it belongs in the output vector, possibly by using extra user-provided information stored in sieve\_data. If the name argument matches the desired criteria, bpsieve should return true. If it does not, bpsieve should return false.

The function bpsieve should be defined in accordance with the typedef:

```
bool (*BPatchFunctionNameSieve) (const char *name, void* sieve_data);
```

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

```
bool findFunction(Dyninst::Address addr,
BPatch_Vectorstd::vector<BPatch_function *> &funcs)
```

Find all functions that have code at the given address, `addr`. Dyninst supports functions that share code, so this method may return more than one `BPatch_function`. These functions are returned via the `funcs` output parameter. This function returns `true` if it finds any functions, `false` otherwise.

```
const BPatch_variableExpr *findVariable(const char *name,
bool showError = true)
const BPatch_variableExpr *findVariable(const BPatch_point
&scope,
const char *name) second form of this method is not implemented on Windows.
```

Performs a lookup and returns a handle to the named variable. The first form of the function looks up only variables of global scope, and the second form uses the passed `BPatch_point` as the scope of the variable. The returned `BPatch_variableExpr` can be used to create references (uses) of the variable in subsequent snippets. The scoping rules used will be those of the source language. If the image was not compiled with debugging symbols, this function will fail even if the variable is defined in the passed scope.

```
const BPatch_type *findType(const char *name)
```

Performs a lookup and returns a handle to the named type. The handle can be used as an argument to `BPatch_addressSpace::malloc` to create new variables of the corresponding type.

```
BPatch_module *findModule(const char *name,
bool substring_match = false)
```

Returns a module named `name` if present in the image. If the match fails, `NULL` is returned. If `substring_match` is `true`, the first module that has `name` as a substring of its name is returned (e.g. to find `libpthread.so.1`, search for `libpthread` with `substring_match` set to `true`).

```
bool getSourceLines(unsigned long addr,
std::vector<std::pair<const char *, unsigned int>
>std::vector<BPatch_statement> & lines)
```

Given an address `addr`, this function returns a vector of pairs of filenames and line numbers at that address. This function is an alias for `BPatch_process::getSourceLines` (see section [4.44.34.3](#)).

Field Code Changed

```
bool getAddressRanges( const char * fileName, unsigned int
lineNo, std::std::vector< std::std::pair< unsigned long,
unsigned long > > & ranges )
```

Given a file name and line number, `fileName` and `lineNo`, this function returns a list of address ranges that this source line was compiled into. This function is an alias for `BPatch_process::getAddressRanges` (see section [4.44.34.3](#)).

```
bool parseNewFunctions( BPatch_Vectorstd::vector<BPatch_module*>
&newModules, const
BPatch_Vectorstd::vector<Dyninst::Address> &funcEntryAddrs)
```

This function takes as input a list of function entry points indicated by the `funcEntryAddrs` vector, which are used to seed parsing in whatever modules they are found. All affected modules are placed in the `newModules` vector, which includes any existing modules in which new functions are found, as well as modules corresponding to new regions of the binary, for which new `BPatch_modules` are created. The return value is `true` in the event that at least one previously unknown function was identified, and `false` otherwise.

#### 4.11 Class BPatch object

An object of this class represents the original executable or a library. It serves as a container of BPatch\_module objects.

```
std::string name()
std::string pathName()
```

Returns the name of this file; either just the file name or the fully path-qualified name.

```
Dyninst::Address fileOffsetToAddr(Dyninst::Offset offset)
```

Converts the provided offset into the file into a full address in memory.

```
struct Region {
    typedef enum { UNKNOWN, CODE, DATA } type t;
    Dyninst::Address base;
    unsigned long size;
    type t type;
};
void regions(std::vector<Region> &regions)
```

Returns information about the address ranges occupied by this object in memory.

```
void modules(std::vector<BPatch_module *> &modules)
```

Returns the modules contained in this object.

Formatted: API



```
std::vector<BPatch function*> *findFunction(  
const char *name,  
std::vector<BPatch function*> &funcs,  
bool showError = true,  
bool regex case sensitive = true,  
bool incUninstrumentable = false)
```

Return a vector of BPatch functions corresponding to name, or NULL if the function does not exist. If name contains a POSIX-extended regular expression, and dont use regex is false, a regular expression search will be performed on function names and matching BPatch functions returned. If showError is true, then Dyninst will report an error via the BPatch::registerErrorCallback if no function is found.

If the incUninstrumentable flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[NOTE: If name is not found to match any demangled function names in the module, the search is repeated as if name is a mangled function name. If this second search succeeds, functions with mangled names matching name are returned instead.]

```
bool findPoints(Dyninst::Address addr,  
std::vector<BPatch point *> &points);
```

Returns a vector of BPatch points that correspond with the provided address, one per function that includes an instruction at that address. There will be one element if there is not overlapping code.

Formatted: Normal

#### **4.104.12 Class BPatch\_module**

An object of this class represents a program module, which is part of a program's executable image. A BPatch\_module represents a source file in an executable or a shared library. Dyninst automatically creates a module called DEFAULT\_MODULE in each executable to hold any objects that it cannot match to a source file. BPatch\_module objects are obtained by calling the BPatch\_image member function getModules.

```
BPatch_Vectorstd::vector<BPatch_function*> *findFunction(  
const char *name,  
BPatch_Vectorstd::vector<BPatch_function*> &funcs,  
bool notify_on_failure = true,  
bool regex_case_sensitive = true,  
bool incUninstrumentable = false)
```

Return a vector of BPatch\_functions matching name, or NULL if the function does not exist. If name contains a POSIX-extended regular expression, a regex search will be performed on function names, and matching BPatch\_functions returned. [NOTE: The ~~BPatch\_Vector~~std::vector argument funcs must be declared fully by the user before

calling this function. Passing in an uninitialized reference will result in undefined behavior.]

Formatted

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[NOTE: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching `name` are returned instead.]

```
BPatch_function *findFunctionByMangled (
    const char *mangled_name,
    bool incUninstrumentable = false)
```

Return a `BPatch_function` for the mangled function `name` defined in the module corresponding to the invoking `BPatch_module`, or `NULL` if it does not define the function.

If the `incUninstrumentable` flag is set, the functions searched will include uninstrumentable functions. The default behavior is to omit these functions.

```
BPatch_function *findFunctionByEntry(Dyninst::Address addr)
```

Formatted: API

Returns the function that begins at the specified address.

```
size_t getAddressWidth()
```

Return the size (in bytes) of a pointer in this module. On 32-bit systems this function will return 4, and on 64-bit systems this function will return 8.

```
bool getSourceLines( unsigned long addr,
    std::vector<BPatch_statementstd::vector<std::pair<const
char *, unsigned int >> & lines )
```

This function returns the line information associated with the mutatee address `addr`. The vector `lines` contain pairs of filenames and line numbers that are associated with `addr`. In many cases only one filename and line number is associated with an address, but certain compiler optimizations may lead to multiple filenames and lines at an address. This information is only available if the mutatee was compiled with debug information.

This function may be more efficient than the `BPatch_process` version of this function. Calling `BPatch_process::getSourceLines` will cause `Dyninst` to parse line information for all modules in a process. If `BPatch_module::getSourceLines` is called then only the debug information in this module will be parsed.

This function returns `true` if it was able to find any line information at `addr`, and `false` otherwise.

```
bool getAddressRanges( char * fileName, unsigned int lineNo,
std::std::vector< std::std::pair< unsigned long, unsigned
long > > & ranges )
```

Given a filename and line number, `fileName` and `lineNo`, this function returns the ranges of mutatee addresses that implement the code range in the output parameter `ranges`. In many cases a source code line will only have one address range implementing it. However, compiler optimizations may turn this into multiple, disjoint address ranges. This information is only available if the mutatee was compiled with debug information.

This function may be more efficient than the `BPatch_process` version of this function. Calling `BPatch_process::getAddressRange` will cause Dyninst to parse line information for all modules in a process. If `BPatch_module::getAddressRange` is called then only the debug information in this module will be parsed.

This function returns `true` if it was able to find any line information, `false` otherwise.

```
const BPatch_Vectorstd::vector<BPatch_function *>
*getProcedures( bool incUninstrumentable = false )
```

Return a vector containing the functions in the module.

```
char *getName(char *buffer, int len)
```

This function copies the filename of the module into `buffer`, up to `len` characters. It returns the value of the `buffer` parameter.

```
char *getFullName(char *buffer, int length)
```

Fills `buffer` with the full path name of a module, up to `length` characters when this information is available.

```
unsigned long getSize()
```

Return the size of the module. The size is defined as the end of the last function minus the start of the first function.

```
bool getVariables(BPatch_Vectorstd::vector<BPatch_variableExpr *>
&vars)
```

Fill the vector `vars` with the global variables that are specified in this module. Returns `false` if no results are found and `true` otherwise.

```
void *getBaseAddr()
```

Return the base address of the module. This address is defined as the start of the first function in the module.

```
bool isSharedLib()
```

This function returns `true` if the module is part of a shared library.

```
BpatchSnippetHandle* insertInitCallback(Bpatch_snippet& callback)
```

This function inserts the snippet `callback` at the entry point of this module's `init` function (creating a new `init` function/section if necessary).

```
BpatchSnippetHandle* insertFiniCallback(Bpatch_snippet& callback)
```

This function inserts the snippet `callback` at the exit point of this module's `fini` function (creating a new `fini` function/section if necessary).

```
const char *getUniqueString(char *buffer, int length)
```

Performs a lookup and returns a unique string for this image. Returns a string that can be compared (via `strcmp`) to indicate if two images refer to the same underlying object file (i.e., executable or library). The contents of the string are implementation specific and defined to have no semantic meaning.

#### 4.14.13 Class `BPatch_snippet`

A snippet is an abstract representation of code to insert into a program. Snippets are defined by creating a new instance of the correct subclass of a snippet. For example, to create a snippet to call a function, create a new instance of the class `BPatch_funcCallExpr`. Creating a snippet does not result in code being inserted into an application. 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 handle to 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.

```
const BPatch_type *getType()
```

Return the type of the snippet. The `BPatch_type` system is described in section [4.14.124.12](#).

Field Code Changed

```
float getCost()
```

Returns an estimate of the number of seconds it would take to execute the snippet. The problems with accurately estimating the cost of executing code are numerous and out of the scope of this document[2]. It is important to realize that the returned cost value is, at best, an estimate.

Field Code Changed

The rest of the classes are derived classes of the class `BPatch_snippet`.

```
BPatch_actualAddressExpr()
```

This snippet results in an expression that evaluates to the actual address of the instrumentation. To access the original address where instrumentation was inserted, use `BPatch_originalAddressExpr`. Note that this actual address is highly dependent on a number of internal variables and has no relation to the original address.

```
BPatch_arithExpr(BPatch_binOp op, const BPatch_snippet &lOperand,
const BPatch_snippet &rOperand)
```

Perform the required binary operation. The available binary operators are:

Operator	Description
<code>BPatch_assign</code>	assign the value of <code>rOperand</code> to <code>lOperand</code>
<code>BPatch_plus</code>	add <code>lOperand</code> and <code>rOperand</code>
<code>BPatch_minus</code>	subtract <code>rOperand</code> from <code>lOperand</code>
<code>BPatch_divide</code>	divide <code>rOperand</code> by <code>lOperand</code>
<code>BPatch_times</code>	multiply <code>rOperand</code> by <code>lOperand</code>
<code>BPatch_ref</code>	Array reference of the form <code>lOperand[rOperand]</code>
<code>BPatch_seq</code>	Define a sequence of two expressions (similar to comma in C)

```
BPatch_arithExpr(BPatch_unOp, const BPatch_snippet &operand)
```

Define a snippet consisting of a unary operator. The unary operators are:

Operator	Description
<code>BPatch_negate</code>	Returns the negation of an integer
<code>BPatch_addr</code>	Returns a pointer to a <code>BPatch_variableExpr</code>
<code>BPatch_deref</code>	Dereferences a pointer

```
BPatch_boolExpr(BPatch_relOp op, const BPatch_snippet &lOperand,
const BPatch_snippet &rOperand)
```

Define a relational snippet. The available operators are:

Operator	Function
<code>BPatch_lt</code>	Return <code>lOperand &lt; rOperand</code>
<code>BPatch_eq</code>	Return <code>lOperand == rOperand</code>
<code>BPatch_gt</code>	Return <code>lOperand &gt; rOperand</code>
<code>BPatch_le</code>	Return <code>lOperand &lt;= rOperand</code>
<code>BPatch_ne</code>	Return <code>lOperand != rOperand</code>
<code>BPatch_ge</code>	Return <code>lOperand &gt;= rOperand</code>
<code>BPatch_and</code>	Return <code>lOperand &amp;&amp; rOperand</code> (Boolean and)
<code>BPatch_or</code>	Return <code>lOperand    rOperand</code> (Boolean or)

The type of the returned snippet is boolean, and the operands are type checked.

`BPatch_breakPointExpr()`

Define a snippet that stops a process when executed by it. The stop can be detected using the `isStopped` member function of `BPatch_process`, and the program's execution can be resumed by calling the `continueExecution` member function of `BPatch_process`.

`BPatch_bytesAccessedExpr()`

This expression returns the number of bytes accessed by a memory operation. For most load/store architecture machines it is a constant expression returning the number of bytes for the particular style of load or store. This snippet is only valid at a memory operation instrumentation point.

`BPatch_constExpr(int value)`

`BPatch_constExpr(long value)`

`BPatch_constExpr(const char *value)`

`BPatch_constExpr(const void *value)`

Define a constant snippet of the appropriate type. The `char*` form of the constructor creates a constant string; the null-terminated string beginning at the location pointed to by the parameter is copied into the application's address space, and the `BPatch_constExpr` that is created refers to the location to which the string was copied.

`BPatch_dynamicTargetExpr()`

This snippet calculates the target of a control flow instruction with a dynamically determined target. It can handle dynamic calls, jumps, and return statements.

`BPatch_effectiveAddressExpr()`

Define an expression that contains the effective address of a memory operation. For a multi-word memory operation (i.e. more than the "natural" operation size of the machine), the effective address is the base address of the operation.

`BPatch_funcCallExpr(const BPatch_function& func,`

`const BPatch_Vectorstd::vector<BPatch_snippet*> &args)`

Define a call to a function. The passed function must be valid for the current code region. `Args` is a list of arguments to pass to the function; the maximum number of arguments varies by platform and is summarized below. If type checking is enabled, the types of the passed arguments are checked against the function to be called. Availability of type checking depends on the source language of the application and program being compiled for debugging.

Platform	Maximum number of arguments
AMD64/EMT-64	No limit
IA-32	No limit
POWER	8 arguments

```
BPatch_funcJumpExpr (const BPatch_function &func)
```

~~Define a snippet that represents a non-returning jump to function `func`. `func` must take the same number and type of arguments as the function in which this snippet is inserted; these arguments will be passed to `func`. `func` must also have the same return type. This snippet can be used to change the implementation of a function, or conditionally change it if the snippet is part of an if statement.~~

~~When `func` returns, control flows as a return from the function in which this snippet is inserted. This snippet has been removed; use `BPatch addressSpace::wrapFunction` instead.~~

Formatted: Font: (Default) Courier New, 10 pt

```
class BPatch_ifExpr(const BPatch_boolExpr &conditional,
    const BPatch_snippet &tClause,
    const BPatch_snippet &fClause)
class BPatch_ifExpr(const BPatch_boolExpr &conditional,
    const BPatch_snippet &tClause)
```

This constructor creates an if statement. The first argument, `conditional`, should be a Boolean expression that will be evaluated to decide which clause should be executed. The second argument, `tClause`, is the snippet to execute if the conditional evaluates to true. The third argument, `fClause`, is the snippet to execute if the conditional evaluates to false. This third argument is optional. Else-if statements, can be constructed by making the `fClause` of an if statement another if statement.

```
BPatch_insnExpr(BPatch_instruction *insn) implemented on x86-64
```

This constructor creates a snippet that allows the user to mimic the effect of an existing instruction. In effect, the snippet “wraps” the instruction and provides a handle to particular components of instruction behavior. This is currently implemented for memory operations, and provides two override methods: `overrideLoadAddress` and `overrideStoreAddress`. Both methods take a `BPatch_snippet` as an argument. Unlike other snippets, this snippet should be installed via a call to `BPatch_process::replaceCode` (to replace the original instruction). For example:

```
// Assume that access is of type BPatch_memoryAccess, as
// provided by a call to BPatch_point->getMemoryAccess. A
// BPatch_memoryAccess is a child of BPatch_instruction, and
// is a valid source of a BPatch_insnExpr.

BPatch_insnExpr insn(access);
```

```
// This example will modify a store by increasing the target
// address by 16.

BPatch_arithExpr newStoreAddr(BPatch_plus,
                              BPatch_effectiveAddressExpr(),
                              BPatch_constExpr(16));

// now override the original store address

insn.overrideStoreAddress(newStoreAddr)

// now replace the original instruction with the new one.
// Point is a BPatch_point corresponding to the desired location, and
// process is a BPatch_process.

process.replaceCode(point, insn);
```

`BPatch_originalAddressExpr()`

This snippet results in an expression that evaluates to the original address of the point where the snippet was inserted. To access the actual address where instrumentation is executed, use `BPatch_actualAddressExpr`.

`BPatch_paramExpr(int paramNum)`

This constructor creates an expression whose value is a parameter being passed to a function. `paramNum` specifies the number of the parameter to return, starting at 0. Since the contents of parameters may change during subroutine execution, this snippet type is only valid at points that are entries to subroutines, or when inserted at a call point with the `when` parameter set to `BPatch_callBefore`.

`BPatch_registerExpr(BPatch_register reg)`  
`BPatch_registerExpr(Dyninst::MachRegister reg)`

This snippet results in an expression whose value is the value in the register at the point of instrumentation.

`BPatch_retExpr()`

This snippet results in an expression that evaluates to the return value of a subroutine. This snippet type is only valid at `BPatch_exit` points, or at a call point with the `when` parameter set to `BPatch_callAfter`.

`BPatch_sequence(const BPatch_Vectorstd::vector<BPatch_snippet*> &items)`

Define a sequence of snippets. The passed snippets will be executed in the order in which they appear in `items`.



```
BPatch_stopThreadExpr(const BPatchStopThreadCallback &callback,  
const BPatch_snippet &calculation)
```

~~The snippet evaluates its calculation snippet and stops the thread that executes it. The result of the calculation snippet is passed up to the mutator, which triggers the callback in the user program. This constructor registers the callback to the stopThreadExpr instance. The same callback may be used for different stopThreadExpr instances. See the definition of BPatchStopThreadCallback in Section 4.1.14.1.1.~~

```
BPatch_threadIndexExpr()
```

This snippet returns an integer expression that contains the thread index of the thread that is executing this snippet. The thread index is the same value that is returned on the mutator side by `BPatch_thread::getBPatchID`.

```
BPatch_tidExpr(const BPatch_process *proc)
```

This snippet results in an integer expression that contains the tid of the thread that is executing this snippet. This can be used to record the threadId, or to filter instrumentation so that it only executes for a specific thread.

```
BPatch_nullExpr()
```

Define a null snippet. This snippet contains no executable statements.

#### **4.124.14 Class BPatch\_type**

The class `BPatch_type` is used to describe the types of variables, parameters, return values, and functions. Instances of the class can represent language predefined types (e.g. int, float), mutatee defined types (e.g., structures compiled into the mutatee application), or mutator defined types (created using the `create*` methods of the `BPatch` class).

```
BPatch_Vector<std::vectorBPatch_Vector::vector<BPatch_field *> *getComponents()
```

Return a vector of the types of the fields in a `BPatch_struct` or `BPatch_union`. If this method is invoked on a type whose `BPatch_dataClass` is not `BPatch_struct` or `BPatch_union`, NULL is returned.

```
BPatch_Vector<std::vectorBPatch_Vector::vector<BPatch_cblock *> *getCblocks()
```

Return the common block classes for the type. The methods of the `BPatch_cblock` can be used to access information about the member of a common block. Since the same named (or anonymous) common block can be defined with different members in different functions, a given common block may have multiple definitions. The vector returned by this function contains one instance of `BPatch_cblock` for each unique definition of the common block. If this method is invoked on a type whose `BPatch_dataClass` is not `BPatch_common`, NULL will be returned.

```
BPatch_type *getConstituentType()
```

Return the type of the base type. For a `BPatch_array` this is the type of each element, for a `BPatch_pointer` this is the type of the object the pointer points to. For `BPatch_typedef` types, this is the original type. For all other types, NULL is returned.

```
enum BPatch_dataClass {
    BPatch_dataScalar,          BPatch_dataEnumerated,
    BPatch_dataTypeClass,      BPatch_dataStructure,
    BPatch_dataUnion,          BPatch_dataArray,
    BPatch_dataPointer,        BPatch_dataReference,
    BPatch_dataFunction,        BPatch_dataTypeAttrib,
    BPatch_dataUnknownType,     BPatch_dataMethod,
    BPatch_dataCommon,          BPatch_dataPrimitive,
    BPatch_dataTypeNumber,      BPatch_dataTypeDefine,
    BPatch_dataNullType }

```

```
BPatch_dataClass getDataClass()
```

Return one of the above data classes for this type.

```
unsigned long getLow()
unsigned long getHigh()
```

Return the upper and lower bound of an array. Calling these two methods on non-array types produces an undefined result.

```
const char *getName()
```

Return the name of the type.

```
bool isCompatible(const BPatch_type &otype)
```

Returns true if `otype` is type compatible with this type. The rules for type compatibility are given in Section [4.274-264-28](#). If the two types are not type compatible, the error reporting callback function will be invoked one or more times with additional information about why the types are not compatible.

Formatted

#### 4.134.15 Class `BPatch_variableExpr`

The `BPatch_variableExpr` class is another class derived from `BPatch_snippet`. It represents a variable or area of memory in a process's address space. A `BPatch_variableExpr` can be obtained from a `BPatch_process` using the `malloc` member function, or from a `BPatch_image` using the `findVariable` member function.

Some `BPatch_variableExpr` have an associated `BPatch_type`, which can be accessed by functions inherited from `BPatch_snippet`. `BPatch_variableExpr` objects will have an associated `BPatch_type` if they originate from binaries with sufficient debug information that describes types, or if they were provided with a `BPatch_type` when created by Dyninst.

**BPatch\_variableExpr** provides several member functions not provided by other types of snippets:

```
void bool readValue(void *dst)
void readValue(void *dst, int size)
```

Read the value of the variable in an application's address space that is represented by this `BPatch_variableExpr`. The `dst` parameter is assumed to point to a buffer large enough to hold a value of the variable's type. If the `size` parameter is supplied, then the number of bytes it specifies will be read. For the first version of this method, if the size of the variable is unknown (i.e., no type information), no data is copied and the method returns false.

```
void bool writeValue(void *src)
void writeValue(void *src, int size)
```

Change the value of the variable in an application's address space that is represented by this `BPatch_variableExpr`. The `src` parameter should point to a value of the variable's type. If the `size` parameter is supplied, then the number of bytes it specifies will be written. For the first version of this method, if the size of the variable is unknown (i.e., no type information), no data is copied and the method returns false.

```
void *getBaseAddr()
```

Return the base address of the variable. This is designed to let users who wish to access elements of arrays or fields in structures do so. It can also be used to obtain the address of a variable to pass a point to that variable as a parameter to a procedure call. It is similar to the ampersand (&) operator in C.

```
BPatch_Vectorstd::vector<BPatch_variableExpr *> *getComponents()
```

Return a [pointer to a](#) vector containing the components of a struct or union. Each element of the vector is one field of the composite type, and contains a variable expression for accessing it.

#### 4.144.16 Class `BPatch_flowGraph`

The `BPatch_flowGraph` class represents the control flow graph of a function. It provides methods for discovering the basic blocks and loops within the function (using which a caller can navigate the graph). A `BPatch_flowGraph` object can be obtained by calling the `getCFG` method of a `BPatch_function` object.

```
bool containsDynamicCallsites()
```

Returns true if the control flow graph contains any dynamic call sites (e.g., calls through a function pointer).

```
void getAllBasicBlocks (BPatch_Set<BPatch_basicBlock*>&)
```

Fill the given set with pointers to all basic blocks in the control flow graph.  
BPatch\_basicBlock is described in section [4.174.174.17](#).

```
void
```

```
getEntryBasicBlock (BPatch_Vectorstd::vector<BPatch_basicBlock*>&)
```

Fill the given vector with pointers to all basic blocks that are entry points to the function.  
BPatch\_basicBlock is described in section [4.174.174.17](#).

```
void
```

```
getExitBasicBlock (BPatch_Vectorstd::vector<BPatch_basicBlock*>&)
```

Fill the given vector with pointers to all basic blocks that are exit points of the function.  
BPatch\_basicBlock is described in section [4.174.174.17](#).

```
void getLoops (BPatch_Vectorstd::vector<BPatch_basicBlockLoop*>&)
```

Fill the given vector with a list of all natural (single entry) loops in the control flow graph.

```
void
```

```
getOuterLoops (BPatch_Vectorstd::vector<BPatch_basicBlockLoop*>&)
```

Fill the given vector with a list of all natural (single entry) outer loops in the control flow graph.

```
BPatch_loopTreeNode *getLoopTree()
```

Return the root node of the tree of loops in this flow graph.

```
enum BPatch_procedureLocation { BPatch_locLoopEntry,  
    BPatch_locLoopExit, BPatch_locLoopStartIter,  
    BPatch_locLoopEndIter }
```

```
BPatch_Vectorstd::vector<BPatch_point*> *findLoopInstPoints(const  
    BPatch_procedureLocation loc, BPatch_basicBlockLoop *loop);
```

Find instrumentation points for the given loop that correspond to the given location: loop entry, loop exit, the start of a loop iteration and the end of a loop iteration.  
BPatch\_locLoopEntry and BPatch\_locLoopExit instrumentation points respectively execute once before the first iteration of a loop and after the last iteration.  
BPatch\_locLoopStartIter and BPatch\_locLoopEndIter respectively execute at the beginning and end of each loop iteration.

[NOTE: Dyninst is not always able to generate a correct flow graph in the presence of indirect jumps. If a function has a case statement or indirect jump instructions, the targets of the jumps are found by searching instruction patterns (peep-hole). The instruction patterns generated are compiler specific and the control flow graph analyses include only the ones we have seen. During the control flow graph generation, if a pattern that is not handled is used for case statement or multi-jump instructions in the function address space, the generated control flow graph may not be complete.]

#### 4.15 Class `BPatch_edge`

The `BPatch_edge` class represents a control flow edge in a `BPatch_flowGraph`.

`BPatch_point *getPoint()`

Return an instrumentation point for this edge. This point can be passed to `BPatch_processor::insertSnippet` to instrument the edge.

enum `BPatch_edgeType` { `CondJumpTaken`, `CondJumpNottaken`, `UncondJump`, `NonJump` }

`BPatch_edgeType getType()`

Return a type describing this edge. A `CondJumpTaken` edge is found after a conditional branch, along the edge that is taken when the condition is true. A `CondJumpNottaken` edge follows the path when the condition is not taken. `UncondJump` is used along an edge that flows out of an unconditional branch that is always taken. `NonJump` is an edge that flows out of a basic block that does not end in a jump, but falls through into the next basic block.

`BPatch_basicBlock *getSource()`

Return the source `BPatch_basicBlock` that this edge flows from.

`BPatch_basicBlock *getTarget()`

Return the target `BPatch_basicBlock` that this edge flows to.

#### 4.16 Class `BPatch_loopTreeNode`

The `BPatch_loopTreeNode` class provides a tree interface to a collection of instances of class `BPatch_basicBlockLoop` contained in a `BPatch_flowGraph`. The structure of the tree

Formatted: APIDesc

~~follows the nesting relationship of the loops in a function's flow graph. Each `BPatch_loopTreeNode` contains a pointer to a loop (represented by `BPatch_basicBlockLoop`), and a set of sub-loops (represented by other `BPatch_loopTreeNode` objects). The root `BPatch_loopTreeNode` instance has a null loop member since a function may contain multiple outer loops. The outer loops are contained in the root instance's vector of children.~~

~~Each instance of `BPatch_loopTreeNode` is given a name that indicates its position in the hierarchy of loops. The name of each root loop takes the form of `loop_*`, where `*` is an integer from 1 to `n`, where `n` is the number of outer loops in the function. Each sub-loop has the name of its parent, followed by a `_y`, where `y` is 1 to `m`, where `m` is the number of sub-loops under the outer loop. For example, consider the following C function:~~

```
void foo() {
    int x, y, z, i;
    for (x=0; x<10; x++) {
        for (y = 0; y<10; y++)
            ...
        for (z = 0; z<10; z++)
            ...
    }
    for (i = 0; i<10; i++) {
        ...
    }
}
```

~~The `foo` function will have a root `BPatch_loopTreeNode`, containing a `NULL` loop entry and two `BPatch_loopTreeNode` children representing the function's outer loops. These children would have names `loop_1` and `loop_2`, respectively representing the `x` and `i` loops. `loop_2` has no children. `loop_1` has two child `BPatch_loopTreeNode` objects, named `loop_1_1` and `loop_1_2`, respectively representing the `y` and `z` loops.~~

~~`BPatch_basicBlockLoop *loop`~~

~~A node in the tree that represents a single `BPatch_basicBlockLoop` instance.~~

~~`BPatch_Vector<BPatch_loopTreeNode *> children`~~

~~The tree nodes for the loops nested under this loop.~~

~~`const char *name()`~~

~~Return a name for this loop that indicates its position in the hierarchy of loops.~~

~~`bool getCallees(BPatch_Vector<BPatch_function *> &v,  
BPatch_process *p)`~~

~~This function fills the vector `v` with the list of functions that are called by this loop.~~

```
const char *getCalleeName(unsigned int i)
```

~~This function return the name of the  $i^{\text{th}}$  function called in the loop's body.~~

```
unsigned int numCallees()
```

~~Returns the number of callees contained in this loop's body.~~

```
BPatch_basicBlockLoop *findLoop(const char *name)
```

~~Finds the loop object for the given canonical loop name.~~

#### 4.17 Class BPatch\_basicBlock

The **BPatch\_basicBlock** class represents a basic block in the application being instrumented. Objects of this class representing the blocks within a function can be obtained using the `BPatch_flowGraph` object for the function. `BPatch_basicBlock` includes methods for navigating through the control flow graph of the containing function.

```
void getSources(BPatch_Vectorstd::vector<BPatch_basicBlock*>&)
```

Fills the given vector with the list of predecessors for this basic block (i.e, basic blocks that have an outgoing edge in the control flow graph leading to this block).

```
void getTargets(BPatch_Vectorstd::vector<BPatch_basicBlock*>&)
```

Fills the given vector with the list of successors for this basic block (i.e, basic blocks that are the destinations of outgoing edges from this block in the control flow graph).

```
void getOutgoingEdges(std::vector<BPatch edge *> &out)
```

Fill out with all of the control flow edges that leave this basic block.

```
void getIncomingEdges(std::vector<BPatch edge *> &inc)
```

Fills inc with all of the control flow edges that point to this basic block.

```
bool getInstructions(std::vector  
<Dyninst::InstructionAPI::Instruction>&)
```

```
bool getInstructions(std::vector <  
std::pair<Dyninst::InstructionAPI::Instruction,  
Address> >&)
```

Fills the given vector with InstructionAPI Instruction objects representing the instructions in this basic block, and returns true if successful. See the InstructionAPI Programmer's Guide for details. The second call also returns the address each instruction starts at.

```
bool dominates(BPatch_basicBlock*)
```

This function returns `true` if the argument is pre-dominated in the control flow graph by this block, and `false` if it is not.

```
BPatch_basicBlock* getImmediateDominator()
```

Return the basic block that immediately pre-dominates this block in the control flow graph.

```
void
```

```
getImmediateDominates(BPatch_Vector<std::vector<BPatch_basicBlock*>&)
```

Fill the given vector with a list of pointers to the basic blocks that are immediately dominated by this basic block in the control flow graph.

```
void getAllDominates(BPatch_Set<BPatch_basicBlock*>&)
```

Fill the given set with pointers to all basic blocks that are dominated by this basic block in the control flow graph.

```
boolvoid
```

```
getSourceBlocks(BPatch_Vector<std::vector<BPatch_sourceBlock*>&)
```

Fill the given vector with pointers to the source blocks contributing to this basic block's instruction sequence.

```
int getBlockNumber()
```

Return the ID number of this basic block. The ID numbers are consecutive from 0 to  $n-1$ , where  $n$  is the number of basic blocks in the flow graph to which this basic block belongs.

```
BPatch_Vector<BPatch_instruction*> getInstructions()
```

~~Return a vector of the instructions that are contained within this basic block.~~

```
bool getInstructions(std::vector<Instruction*>&)  
implemented for IA32, AMD64, and POWER
```

~~Fills the given vector with InstructionAPI Instruction objects representing the instructions in this basic block, and returns true if successful. See the InstructionAPI Programmer's Guide for details.~~

```
void getIncomingEdges(BPatch_Vector<BPatch_edge*> &in)
```

~~Fills in with all of the control flow edges that point to this basic block.~~

```
BPatch_Vector<std::vector<BPatch_point*> findPoint(const  
BPatch_Set<BPatch_opCode> &ops)
```

Find all points in the basic block that match the given operation.

```
BPatch_point *findEntryPoint()  
BPatch_point *findExitPoint()
```

Find the entry or exit point of the block.~~void getOutgoingEdges(BPatch\_Vector<BPatch\_edge\*> &out)~~

*dyninstAPI*

Formatted: Indent: Left: 1", First line: 0.5"

Formatted: Font: Courier New, 12 pt, Not Italic

Formatted: API



~~Fill out with all of the control flow edges that leave this basic block.~~

```
unsigned long getStartAddress()
```

This function returns the starting address of the basic block. The address returned is an absolute address.

```
unsigned long getEndAddress()
```

This function returns the end address of the basic block. The address returned is an absolute address.

```
unsigned long getLastInsnAddress()
```

Return the address of the last instruction in a basic block.

```
bool isEntryBlock()
```

This function returns `true` if this basic block is an entry block into a function.

```
bool isExitBlock()
```

This function returns `true` if this basic block is an exit block of a function.

```
unsigned size()
```

Return the size of a basic block. The size is defined as the difference between the end address and the start address of the basic block.

#### 4.18 Class BPatch edge

The BPatch edge class represents a control flow edge in a BPatch flowGraph.

```
BPatch point *getPoint()
```

Return an instrumentation point for this edge. This point can be passed to BPatch process::insertSnippet to instrument the edge.

```
enum BPatch edgeType { CondJumpTaken, CondJumpNottaken,  
UncondJump, NonJump }
```

```
BPatch edgeType getType()
```

Return a type describing this edge. A CondJumpTaken edge is found after a conditional branch, along the edge that is taken when the condition is true. A CondJumpNottaken edge follows the path when the condition is not taken. UncondJump is used along an edge that flows out of an unconditional branch that is always taken. NonJump is an edge that flows out of a basic block that does not end in a jump, but falls through into the next basic block.

BPatch\_basicBlock \*getSource()

Return the source BPatch\_basicBlock that this edge flows from.

BPatch\_basicBlock \*getTarget()

Return the target BPatch\_basicBlock that this edge flows to.

BPatch\_flowGraph \*getFlowGraph()

Returns the CFG that contains the edge.

#### **4.184.19 Class BPatch\_basicBlockLoop**

An object of this class represents a loop in the code of the application being instrumented.

bool containsAddress(unsigned long addr)

Returns true if addr is contained within any of the basic blocks that compose this loop, excluding the block of any of its sub-loops.

bool containsAddressInclusive(unsigned long addr)

Returns true if addr is contained within any of the basic blocks that compose this loop, or in the blocks of any of its sub-loops.

BPatch\_edge \*getBackEdge()

Return a pointer to a back edge that defines this natural loop.

int getBackEdges(~~BPatch\_Vector~~std::vector<BPatch\_edge \*> &edges)

Returns the number of back edges that define this natural loop and adds those edges to the edges vector. A loop can have multiple back edges if those edges define loops comprising the same set of nodes.

~~bool~~void

getContainedLoops(~~BPatch\_Vector~~std::vector<BPatch\_basicBlockLoop\*>&)

Fill the given vector with a list of the loops nested within this loop.

BPatch\_flowGraph \*getFlowGraph()

Return a pointer to the control flow graph that contains this loop.

BPatch\_basicBlock \*getLoopHead()

Return a pointer to the basic block that is at the head of this loop.

```
boolvoid
  getOuterLoops (BPatch_Vectorstd::vector<BPatch_basicBlockLoop
    *>&)
```

Fill the given vector with a list of the outer loops nested within this loop.

```
boolvoid
  getLoopBasicBlocks (BPatch_Vectorstd::vector<BPatch_basicBlock
    *>&)
```

Fill the given vector with a list of all basic blocks that are part of this loop.

```
boolvoid getLoopBasicBlocksExclusive (
  BPatch_Vectorstd::vector<BPatch_basicBlock*>&)
```

Fill the given vector with a list of all basic blocks that are part of this loop but not its sub-loops.

```
BPatch_basicBlock* getLoopHead()
```

Return the basic block at the head of this loop.

```
bool hasAncestor (BPatch_basicBlockLoop*)
```

Returns true if this loop is nested within the given loop (the given loop is one of its ancestors in the tree of loops).

```
bool hasBlock (BPatch_basicBlock *b)
```

Returns true if this loop or any of its sub-loops contain b, false otherwise.

```
bool hasBlockExclusive (BPatch_basicBlock *b)
```

Returns true if this loop, excluding its sub-loops, contains b, false otherwise.

#### 4.20 Class BPatch\_loopTreeNode

The BPatch\_loopTreeNode class provides a tree interface to a collection of instances of class BPatch\_basicBlockLoop contained in a BPatch\_flowGraph. The structure of the tree follows the nesting relationship of the loops in a function's flow graph. Each BPatch\_loopTreeNode contains a pointer to a loop (represented by BPatch\_basicBlockLoop), and a set of sub-loops (represented by other BPatch\_loopTreeNode objects). The root BPatch\_loopTreeNode instance has a null loop member since a function may contain multiple outer loops. The outer loops are contained in the root instance's vector of children.

Each instance of BPatch\_loopTreeNode is given a name that indicates its position in the hierarchy of loops. The name of each root loop takes the form of loop\_x, where x is an integer from 1

to  $n$ , where  $n$  is the number of outer loops in the function. Each sub-loop has the name of its parent, followed by a  $_v$ , where  $v$  is 1 to  $m$ , where  $m$  is the number of sub-loops under the outer loop. For example, consider the following C function:

```
void foo() {
    int x, y, z, i;
    for (x=0; x<10; x++) {
        for (y = 0; y<10; y++)
            ...
        for (z = 0; z<10; z++)
            ...
    }
    for (i = 0; i<10; i++) {
        ...
    }
}
```

The `foo` function will have a root `BPatch_loopTreeNode`, containing a `NULL` `loop` entry and two `BPatch_loopTreeNode` children representing the function's outer loops. These children would have names `loop_1` and `loop_2`, respectively representing the `x` and `i` loops. `loop_2` has no children. `loop_1` has two child `BPatch_loopTreeNode` objects, named `loop_1_1` and `loop_1_2`, respectively representing the `y` and `z` loops.

`BPatch_basicBlockLoop *loop`

A node in the tree that represents a single `BPatch_basicBlockLoop` instance.

`std::vector<BPatch_loopTreeNode *> children`

The tree nodes for the loops nested under this loop.

`const char *name()`

Return a name for this loop that indicates its position in the hierarchy of loops.

`bool getCallees(std::vector<BPatch_function *> &v,`  
`BPatch addressSpace *p)`

This function fills the vector `v` with the list of functions that are called by this loop.

`const char *getCalleeName(unsigned int i)`

This function return the name of the  $i^{\text{th}}$  function called in the loop's body.

`unsigned int numCallees()`

Returns the number of callees contained in this loop's body.

`BPatch_basicBlockLoop *findLoop(const char *name)`

Finds the loop object for the given canonical loop name.

#### ~~4.19~~ **Class BPatch\_instruction**

~~A **BPatch\_instruction** represents a single machine instruction in the `BPatch_flowGraph`. **BPatch\_instructions** can be retrieved with the `BPatch_basicBlock::getInstructions` call.~~

~~`void *getAddress()`~~

~~———— This function returns the starting address of this instruction. This function returns an address in the mutatee, not in the mutator.~~

~~`BPatch_point *getInstPoint()`~~

~~———— This function returns a `BPatch_point` at this instruction. This point can be passed to `BPatch_process::insertSnippet` to instrument this instruction.~~

#### ~~4.20~~4.21 **Class BPatch\_register**

A **BPatch\_register** represents a single register of the mutatee. The list of `BPatch_registers` can be retrieved with the `BPatch_addressSpace::getRegisters` method.

~~`std::string *name()`~~

This function returns the canonical name of the register.

#### ~~4.21~~4.22 **Class BPatch\_sourceBlock**

An object of this class represents a source code level block. Each source block objects consists of a source file and a set of source lines in that source file. This class is used to fill source line information for each basic block in the control flow graph. For each basic block in the control flow graph there is one or more source block object(s) that correspond to the source files and their lines contributing to the instruction sequence of the basic block.

`const char* getSourceFile()`

Returns a pointer to the name of the source file in which this source block occurs.

`void getSourceLines(BPatch_Vectorstd::vector<unsigned short>&)`

Fill the given vector with a list of the lines contained within this source block.

#### ~~4.22~~4.23 **Class BPatch\_cblock**

This class is used to access information about a common block.

~~BPatch\_Vector~~std::vector<BPatch\_field \*> \*getComponents()

Return a vector containing the individual variables of the common block.

~~BPatch\_Vector~~std::vector<BPatch\_function \*> \*getFunctions()

Return a vector of the functions that can see this common block with the set of fields described in `getComponents`. However, other functions that define this common block with a different set of variables (or sizes of any variable) will not be returned.

#### 4.234.24 Class BPatch\_frame

A **BPatch\_frame** object represents a stack frame. The `getCallStack` member function of `BPatch_thread` returns a vector of `BPatch_frame` objects representing the frames currently on the stack.

BPatch\_frameType getFrameType()

Return the type of the stack frame. Possible types are:

Frame Type	Meaning
BPatch_frameNormal	A normal stack frame.
BPatch_frameSignal	A frame that represents a signal invocation.
BPatch_frameTrampoline	A frame the represents a call into instrumentation code.

void \*getFP()

Return the frame pointer for the stack frame.

void \*getPC()

Returns the program counter associated with the stack frame.

BPatch\_function \*findFunction()

Returns the function associated with the stack frame.

#### 4.244.25 Container Classes

##### 4.24.14.25.1 Class ~~BPatch\_Vector~~std::vector

The ~~BPatch\_Vector~~std::vector class is a container used to hold other objects used by the API. As of Dyninst 5.0, ~~BPatch\_Vector~~std::vector is an alias for the C++ Standard Template Library (STL) ~~std::~~std::vector.

#### 4.24.24.25.2 Class BPatch\_Set

**BPatch\_Set** is another container class, similar to the set class in the STL. ~~In addition the methods provided by `std::set`, it provides the following compatibility methods: It maintains a collection of objects and provides fast lookup. Elements are ordered by a comparison function, which can be user supplied. This allows for efficiently returning a sorted list of elements, or returning the value of the minimum or maximum element.~~

Formatted: Font: (Default) Courier New, 10 pt

`BPatch_Set()`

A constructor that creates an empty set with the default comparison function.

`BPatch_Set(const BPatch_Set<T, Compare>& newBPatch_Set)`

Copy constructor.

~~`int size()`~~

~~Return the number of elements in the set.~~

~~`bool empty()`~~

~~Returns true if the set is empty, or false if it is not.~~

~~`void insert(const T&)`~~

~~Insert the given element into the set.~~

`void remove(const T&)`

Remove the given element from the set.

`bool contains(const T&)`

Returns true if the argument is a member of the set, otherwise returns false.

`T* elements(T*)`

~~`void elements(std::vector<T> &)`~~

Formatted: API

Fill an array (or vector) with a list of the elements in the set that are sorted in ascending order according to the comparison function. The input argument should point to an array large enough to hold the elements. This function returns its input argument, unless the set is empty, in which case it returns `NULL`.

`T minimum()`

Return the minimum element in the set, as determined by the comparison function. For an empty set, the result is undefined.

`T maximum()`

Return the maximum element in the set, as determined by the comparison function. For an empty set, the result is undefined.

~~BPatch\_Set<T, Compare>& operator= (const BPatch\_Set<T, Compare>&)~~

~~the assignment operator.~~

~~bool operator== (const BPatch\_Set<T, Compare>&)~~

~~The equality operator. Returns true if both sets consist entirely of elements that are each equal to an element in the other set, or if both sets are empty.~~

~~bool operator!= (const BPatch\_Set<T, Compare>&)~~

~~The inequality operator. Returns true if either set contains an element not in the other set.~~

BPatch\_Set<T, Compare>& operator+= (const T&)

Add the given object to the set.

BPatch\_Set<T, Compare>& operator|= (const BPatch\_Set<T, Compare>&)

Set union operator. Assign the result of the union to the set on the left hand side.

BPatch\_Set<T, Compare>& operator&= (const BPatch\_Set<T, Compare>&)

Set intersection operator. Assign the result of the intersection to the set on the left hand side.

BPatch\_Set<T, Compare>& operator-= (const BPatch\_Set<T, Compare>&)

Set difference operator. Assign the difference of the sets to the set on the left hand side.

BPatch\_Set<T, Compare> operator| (const BPatch\_Set<T, Compare>&)

Set union operator.

BPatch\_Set<T, Compare> operator& (const BPatch\_Set<T, Compare>&)

Set intersection operator.

BPatch\_Set<T, Compare> operator- (const BPatch\_Set<T, Compare>&)

Set difference operator.

#### 4.254.26 Memory Access Classes

Instrumentation points created through findPoint(const BPatch\_Set<BPatch\_opCode>& ops) get memory access information attached to them. This information is used by the memory access snippets, but is also available to the API user. The classes that encapsulate memory access information are contained in the BPatch\_memoryAccess\_NP.h header.



#### **4.25.14.26.1 Class BPatch\_memoryAccess**

This class encapsulates a memory access abstraction. It contains information that describes the memory access type: read, write, read/write, or prefetch. It also contains information that allows the effective address and the number of bytes transferred to be determined.

```
bool isALoad_NP()
```

Returns `true` if the memory access is a load (memory is read into a register).

```
bool isASore_NP()
```

Returns `true` if the memory access is write. Some machine instructions may both load and store.

```
bool isAPrefetch_NP()
```

Returns `true` if memory access is a prefetch (i.e, it has no observable effect on user registers). If this returns `true`, the instruction is considered neither load nor store. *Prefetches are detected only on IA32.*

```
short prefetchType_NP()
```

If the memory access is a prefetch, this method returns a platform specific prefetch type.

```
BPatch_addrSpec_NP getStartAddr_NP()
```

Return an address specification that allows the effective address of a memory reference to be computed. For example, on the x86 platform a memory access instruction operand may contain a base register, an index register, a scaling value, and a constant base. The `BPatch_addrSpec_NP` describes each of these values.

```
BPatch_countSpec_NP getByteCount_NP()
```

Return a specification that describes the number of bytes transferred by the memory access.

#### **4.25.24.26.2 Class BPatch\_addrSpec\_NP**

This class encapsulates the information required to determine an effective address at runtime. The general representation for an address is a sum of two registers and a constant; this may change in future releases. Some architectures use only certain bits of a register (e.g. bits 25:31 of XER register on the Power chip family); these are represented as pseudo-registers. The numbering scheme for registers and pseudo-registers is implementation dependent and should not be relied upon; it may change in future releases.

```
int getImm()
```

Return the constant offset. This may be positive or negative.

```
int getReg(unsigned i)
```

Return the register number for the  $i^{\text{th}}$  register in the sum, where  $0 \leq i \leq 2$ . Register numbers are positive; a value of -1 means no register.

```
int getScale()
```

Returns any scaling factor used in the memory address computation.

#### **4.25.34.26.3 Class BPatch\_countSpec\_NP**

This class encapsulates the information required to determine the number of bytes transferred by a memory access.

#### **4.264.27 Type System**

The Dyninst type system is based on the notion of structural equivalence. Structural equivalence was selected to allow the system the greatest flexibility in allowing users to write mutators that work with applications compiled both with and without debugging symbols enabled. Using the `create*` methods of the `BPatch` class, a mutator can construct type definitions for existing mutatee structures. This information allows a mutator to read and write complex types even if the application program has been compiled without debugging information. However, if the application has been compiled with debugging information, Dyninst will verify the type compatibility of the operations performed by the mutator.

The rules for type computability are that two types must be of the same storage class (i.e. arrays are only compatible with other arrays) to be type compatible. For each storage class, the following additional requirements must be met for two types to be compatible:

`BPatch_dataScalar`

Scalars are compatible if their names are the same (as defined by `strcmp`) and their sizes are the same.

`BPatch_dataPointer`

Pointers are compatible if the types they point to are compatible.

`BPatch_dataFunc`

Functions are compatible if their return types are compatible, they have same number of parameters, and position by position each element of the parameter list is type compatible.

`BPatch_dataArray`

Arrays are compatible if they have the same number of elements (regardless of their lower and upper bounds) and the base element types are type compatible.

BPatch\_dataEnumerated

Enumerated types are compatible if they have the same number of elements and the identifiers of the elements are the same.

BPatch\_dataStructure

BPatch\_dataUnion

Structures and unions are compatible if they have the same number of constituent parts (fields) and item by item each field is type compatible with the corresponds field of the other type.

In addition, if either of the types is the type BPatch\_unknownType, then the two types are compatible. Variables in mutatee programs that have not been compiled with debugging symbols (or in the symbols are in a format that the Dyninst library does not recognize) will be of type BPatch\_unknownType.

#### 4.27 Hybrid Code Analysis

~~Activating hybrid code analysis causes Dyninst to augment its static analysis of the code with run time code discovery techniques. Hybrid analysis is useful even for binaries that have not been deliberately obfuscated, as Dyninst's static analysis of the program binary may miss code for which there are no function symbols, and may parse functions incompletely if they contain indirect jumps whose targets are not recognizable as jump table entries. This mechanism also allows for the analysis of analysis-resistant binaries that are stripped of symbol information and may dynamically unpack binary code or modify existing code. These use cases have corresponding modes of hybrid code analysis:~~

- ~~1. **Instrumentation-based discovery:** This mode makes no modification to Dyninst's static analysis, except to mark statically unresolved control transfers for instrumentation. The following types of control transfers are instrumented: indirect jump instructions that do not follow jump table conventions, indirect calls with statically unresolved targets, and static control transfers to invalid or un-initialized memory regions.~~
- ~~2. **Malware-mode discovery:** This mode makes Dyninst's static analysis more conservative, and compensates through heavier use of instrumentation. It also assumes that the binary may attempt to modify its own code, so it write-protects all memory pages that contain code, causing overwrite attempts to raise signals. This mode can only be activated if the user registers code discovery and code overwrite callbacks, in which case the user can specifically request that this mode be activated, or rely on Dyninst's heuristics for detecting the likely presence of code obfuscations.~~
- ~~3. **Static analysis only:** This is Dyninst's traditional mode of analysis; no run-time code discovery is performed.~~

**Heuristic analysis mode selection.** The user can set the desired hybrid analysis mode or rely on Dyninst's heuristics. Dyninst chooses static analysis only for binaries with symbol information. Instrumentation-based discovery is the default for stripped binaries, but malware mode is activated if one of the following conditions is met:

1. The binary lacks a .text section and its first section is writable
2. The binary's .text section is writable or uninitialized
3. The program contains static control transfers to un-initialized regions
4. The binary contains contains control-transfer obfuscations

```
enum BPatch_hybridMode { BPatch_staticMode, BPatch_instMode,
    BPatch_malwareMode };
bool BPatch_Image::setHybridMode(BPatch_hybridMode mode);
```

Set the hybrid analysis mode to static analysis only (turns off heuristic enabling of hybrid modes), set it to do instrumentation-based code discovery only, or set it to malware mode, in which the program is assumed to be analysis-resistant.

```
static void BPatchCodeDiscoveryCallback
(std::vector<BPatch_function*> &newFunes,
 std::vector<BPatch_function*> &modifiedFunes);
```

This callback is invoked whenever previously un-analyzed code is discovered through runtime analysis, and delivers a vector of functions whose analysis has been modified and a vector of functions that are newly discovered.

```
static void BPatchCodeOverwriteBeginCallback
(std::vector<BPatch_basicBlock*> &overwriteLoopBlocks);
```

This callback allows the user to remove any instrumentation as the program starts writing to a code page, which may be desirable as instrumentation cannot be removed during the overwrite loop's execution, and any breakpoint instrumentation will dramatically slow the loop's execution. Only invoked if hybrid analysis mode is set to `BPatch_malwareMode`.

```
static void BPatchCodeOverwriteEndCallback(std::vector<Address>
&deadFuncAdrs, std::vector<BPatch_function*>
&modifiedFunes, std::vector<BPatch_function*> &newFunes);
```

This callback delivers the effects of the overwrite loop when it is done executing. In many cases no code will have changed. This function is only called if the Dyninst's hybrid analysis mode is set to `BPatch_malwareMode`.

~~Execute at the signal handler's entry point. If the handler shifts execution to un-analyzed code, this will cause the `BPatchCodeDiscoveryCallback` to be invoked.~~

~~`BPatch_process::removeDebuggingArtifacts()`~~

~~Remove debugging artifacts. This function invoked automatically if hybrid analysis mode is set to `BPatch_malwareMode`, but there is no harm in calling it twice.~~

~~`bool BPatch_flowGraph::isValid()`~~

~~When a function's analysis is updated, its `BPatch_function` object remains valid, but its `BPatch_flowGraph` does not. A new CFG can be obtained from the `BPatch_function` when the CFG is found to be invalid.~~

~~enum {BPatch\_noInterp, BPatch\_interpAsTarget,  
BPatch\_interpAsReturnAddr} BPatch\_stInterpret;  
`BPatch_stopThreadExpr` (`BPatch_snippet *calculation`,  
`BPatch_stInterpret interp = BPatch_noInterp`, `bool useCache = false`)~~

~~Interpret the calculation as an address, which will cause addresses corresponding to relocated code and Dyninst's instrumentation to be translated into an unrelocated address.~~

|

## 5. USING DYNINSTAPI WITH THE COMPONENT LIBRARIES

In this section, we describe how to access the underlying component library abstractions from corresponding Dyninst abstractions. The component libraries (SymtabAPI, InstructionAPI, ParseAPI, and PatchAPI) often provide greater functionality and cleaner interfaces than Dyninst, and thus users may wish to use a mix of abstractions. In general, users may access component library abstractions via a `convert` function, which is overloaded and namespaced to give consistent behavior. The definitions of all component library abstractions are located in the appropriate documentation.

```
PatchAPI::PatchMgrPtr PatchAPI::convert(BPatch addressSpace *);

PatchAPI::PatchObject *PatchAPI::convert(BPatch object *);
ParseAPI::CodeObject *ParseAPI::convert(BPatch object *);
SymtabAPI::Symtab *SymtabAPI::convert(BPatch object *);

SymtabAPI::Module *SymtabAPI::convert(BPatch module *);

PatchAPI::PatchFunction *PatchAPI::convert(BPatch function *);
ParseAPI::Function *ParseAPI::convert(BPatch function *);

PatchAPI::PatchBlock *PatchAPI::convert(BPatch basicBlock *);
ParseAPI::Block *ParseAPI::convert(BPatch basicBlock *);

PatchAPI::PatchEdge *PatchAPI::convert(BPatch edge *);
ParseAPI::Edge *ParseAPI::convert(BPatch edge *);

PatchAPI::Point *PatchAPI::convert(BPatch point *, BPatch callWhen);
PatchAPI::SnippetPtr PatchAPI::convert(BPatch snippet *);

SymtabAPI::Type *SymtabAPI::convert(BPatch type *);
```

### 5.6. USING THE API

In this section, we describe the steps needed to compile your mutator and mutatee programs and to run them. First we give you an overview of the major steps and then we explain each one in detail.

#### 5.6.1 Overview of Major Steps

To use Dyninst, you have to:

(1) *Build and install DyninstAPI (Section 6.25.2):* You will need to build and install the DyninstAPI library.

(2) *Create a mutator program (Section 6.35.3-6.35.4):* You need to create a program that will modify some other program. For an example, see the mutator shown in Appendix A - Appendix A

Formatted: Body

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Code, Indent: First line: 0.2"

Formatted: Code

Formatted: Code

Formatted

Formatted: Body

Formatted: Numbered + Level: 1 +  
Numbering Style: 1, 2, 3, ... + Start at: 1 +  
Alignment: Left + Aligned at: 0" + Indent at:  
0.25"

Field Code Changed

Formatted

Formatted

Formatted: Font: Not Italic

Formatted: Indent: Left: 0.25", No bullets or  
numbering

Formatted: Font: 12 pt

Field Code Changed

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Numbered + Level: 1 +  
Numbering Style: 1, 2, 3, ... + Start at: 1 +  
Alignment: Left + Aligned at: 0" + Indent at:  
0.25"

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: 12 pt

~~(1)~~(3) *Set up the mutatee* (Section [6.45.45.3](#)): On some platforms, you need to link your application with Dyninst's run time instrumentation library. [NOTE: This step is only needed in the current release of the API. Future releases will eliminate this restriction.]

~~(2)~~(4) *Run the mutator* (Section [6.55.55.4](#)): The mutator will either create a new process or attach to an existing one (depending on the whether createProcess or attachProcess is used).

Sections [6.25.25.2](#) through [6.55.55.4](#) explain these steps in more detail. ~~In this section, we assume that you have already installed the API distribution and setup the PLATFORM and DYNINST\_ROOT environment variables. The installation of the API is described in the README file in the distribution tar file.~~

## 6.2 Building and Installing DyninstAPI

This section describes how to build and install Dyninst, which can be downloaded from <http://www.dyninst.org>. You may either download source code or an installation package; if you choose to use an installation package, you should run the installation package and then skip to section [6.35.3](#).

### 6.2.1 Building on UNIX

Building Dyninst on UNIX platforms is a four step process that involves: unpacking the Dyninst source, installing any Dyninst dependencies, configuring paths in make.config.local, and running the build.

Dyninst's source code is packaged in a tar.gz format. If your Dyninst source tarball is called `srcDist_v8.0.tar.gz`, then you could extract it with the command `gunzip srcDist_v8.0.tar.gz ; tar -xvf srcDist_v8.0.tar`. This will create two directories: `dyninst` and `scripts`.

Dyninst has several dependencies, including libdwarf, libelf, and boost, depending on what platform you are using, which must be installed before Dyninst can be built. Note that for most of these packages Dyninst needs to be able to access the package's include files, which means that development versions are required. If a version number is listed for a package, then there are known bugs that may affect Dyninst with earlier versions of the package.

<a href="#">Linux/x86</a>	<a href="#">libdwarf-20100808</a>
	<a href="#">libelf</a>
<a href="#">Linux/x86-64</a>	<a href="#">Libdwarf-20100808</a>
	<a href="#">libelf</a>
<a href="#">Linux/Power</a>	<a href="#">libdwarf-20100808</a>
	<a href="#">libelf</a>

Field Code Changed

Field Code Changed

Formatted: Font: Bold

Field Code Changed

Formatted: Heading 3, H3

Formatted Table



At the time of this writing the Linux packages could be found at:

- [libdwarf - http://reality.sgiweb.org/davea/dwarf.html](http://reality.sgiweb.org/davea/dwarf.html)
- [libelf - http://www.mr511.de/software/english.html](http://www.mr511.de/software/english.html)
- [libunwind - http://www.hpl.hp.com/research/linux/libunwind/download.php4](http://www.hpl.hp.com/research/linux/libunwind/download.php4)
- 

Once the dependencies for Dyninst have been installed, Dyninst must be configured to know where to find these packages. This can be done with the configure script located in the root of the Dyninst source directory; run `./configure --help` for a current list of options.

~~This is done through Dyninst's `dyninst/make.config.local` file. This file must be written in GNU Makefile syntax and must specify directory locations for each dependency. Specifically, `LIBDWARFDIR`, `LIBELFDIR` and `TCLTK_DIR` variables must be set. `LIBDWARFDIR` should be set to the absolute path of libdwarf library where `dwarf.h` and `libdwarf.h` files reside. `LIBELFDIR` should be set to the absolute path where `libelf.a` and `libelf.so` files are located.~~

~~`make.config.local` can also be generated via `autoconf`. On most systems, `configure` with no arguments will generate a valid `make.config.local`. For more details, `configure --help` will provide a full list of configuration options.~~

~~The next thing is to set `DYNINST_ROOT` and `LD_LIBRARY_PATH` environment variables. `DYNINST_ROOT` should be set to path of the directory that contains `dyninst` and `scripts` subdirectories.~~

~~The `LD_LIBRARY_PATH` variable should be set in a way that it includes `libdwarf home directory/lib` and, if configuration options were used, the location specified by `--exec-prefix`.~~

~~Once configuration is complete execute `make install` to build Dyninst and install it to the directory you specified. Once `make.config.local` is set you are ready to build Dyninst. Change to the `dyninst` directory and execute the command `make DyninstAPI`. This will build Dyninst's mutator library, the Dyninst runtime library, and Dyninst's test suite. Successfully built binaries will be stored in a subdirectory of the `dyninstAPI` directory based on your current platform. To install the mutator library, execute the command `make install DyninstAPI`. This will copy header files and built binaries to the default location for your system, or, if configuration options were used, to paths specified by `--prefix` and `--exec-prefix`.~~

## 6.2.2 Building on Windows

Dyninst for Windows is built with Microsoft Visual Studio 2008<sup>10</sup> project and solution files. Building Dyninst for Windows is similar to UNIX in that it is a four step process: Unpack the DyninstAPI source code, install Dyninst's package dependencies, configure Visual Studio to use the dependencies, and run the build system.

*dyninstAPI*

Field Code Changed

Field Code Changed

Field Code Changed

Formatted: Bulleted + Level: 1 + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

Formatted: Font: (Default) Courier New, 10 pt

Formatted

Formatted

Formatted

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: 12 pt

Formatted: Font: (Default) Courier New, 10 pt

Formatted

Formatted

Formatted

Formatted

Formatted

Formatted: Font: 12 pt

Formatted: Heading 3,H3

Dyninst source code is distributed as part of a tar.gz package. Most popular unzipping programs are capable of handling this format. Extracting the Dyninst tarball results in two directories: dyninst and scripts.

Dyninst for Windows depends on Microsoft's Debugging Tools for Windows, which could be found at <http://www.microsoft.com/whdc/devtools/debugging/default.msp> at the time of this writing. Download these tools and install them at an appropriate location. Make sure to do a custom install and install the SDK, which is not always installed by default. For the rest of this section, we will assume that the Debugging Tools are installed at `c:\program files\Debugging Tools for Windows`. If this is not the case, then adjust the following instruction appropriately.

Once the Debugging Tools are installed, Visual Studio must be configured to use them. We need to add the Debugging Tools include and library directories to Visual Studio's search paths. In Visual Studio 2008 select Options... from the tools menu. Next select Projects and VC++ Directories from the pane on the left. You should see a list of directories that are sorted into categories such as 'Executable files', 'Include files', etc. The current category can be changed with a drop down box in the upper right hand side of the Dialog.

First, change to the 'Library files' category, and add an entry that points to `C:\Program Files\Debugging Tools for Windows\sdk\lib\i386`. Make sure that this entry is above Visual Studio's default search paths.

Next, Change to the 'Include files' category and make a new entry in the list that points to `C:\Program Files\Debugging Tools for Windows\sdk\inc`. Also make sure that this entry is above Visual Studio's default search paths. Some users have had a problem where Visual Studio cannot find the `eveconst.h` file. You may need to add the directory containing this file to the include search path. We have seen it installed at `$(VCInstallDir)/../Visual Studio SDKs/DIA SDK/include`, although you may need to search for it.

Once you have installed and configured the Debugging Tools for Windows you are ready to build Dyninst. First, you need to create the directories where Dyninst will install its completed build. From the dyninst directory you need to create the directories `../i386-unknown-nt4.0/bin` and `../i386-unknown-nt4.0/lib`. Next open the solution file `dyninst/DyninstAPI.sln` with Visual Studio. You can then build Dyninst by select 'Build Solution' from the build menu. This will build the Dyninst mutator library, the runtime library, and the test suite.

### 5.26.3 Creating a Mutator Program

The first step in using Dyninst is to create a mutator program. The mutator program specifies the mutatee (either by naming an executable to start or by supplying a process ID for an existing pro-

*dyninstAPI*

Field Code Changed

Formatted

Formatted: Normal

cess). In addition, your mutator will include the calls to the API library to modify the mutatee. For the rest of this section, we assume that the mutator is the sample program given in [Appendix A - Complete Examples](#). The following fragment of a Makefile shows how to link your mutator program with the Dyninst library on most platforms:

```
# DYNINST INCLUDE and DYNINST LIB should be set to locations
# where Dyninst header and library files were installed, respectively

retee.o: retee.c
$(CC) -c $(CFLAGS) -I$(DYNINST_ROOT)$(DYNINST_INCLUDE)/dyninst/dyninstAPI/h \
retee.c

retee: retee.o
$(CC) retee.o -L$(DYNINST_ROOT)/lib/$(PLATFORM)/lib \
-lldyninstAPI -ldyninstAPI -o retee
```

On Linux, the options `-lself` and `-ldwarf` must be added to the link step. On AIX, the option `-lbsd` must also be added to the link step. You will also need to make sure that the `LD_LIBRARY_PATH` or `LIBPATH (AIX)` environment variable includes the directory that contains the Dyninst shared library. This is typically `$(DYNINST_ROOT)/lib/$(PLATFORM)`. Finally, on AIX, Dyninst mutators can easily exceed the 256MB of data space that is available to the default memory model. Your mutator can be re-linked as follows to work around this problem:

```
ldedit bmaxdata:0x80000000/dsa MUTATOR_EXECUTABLE
```

Some of these libraries, such as `libdwarf` and `libelf`, may not be standard on various platforms. Check the README file in `dyninst/dyninstAPI` for more information on where to find these libraries.

Under Windows NT, the mutator also needs to be linked with the `dbghelp` library, which is included in the Microsoft Platform SDK. Below is a fragment from a Makefile for Windows NT:

```
# DYNINST INCLUDE and DYNINST LIB should be set to locations
# where Dyninst header and library files were installed, respectively

CC = cl

retee.obj: retee.c
$(CC) -c $(CFLAGS) -I$(DYNINST_ROOT)/dyninst/dyninstAPI/h \
```

```

retee.exe: retee.obj
link -out:retee.exe retee.obj \
-$(DYNINST_ROOT)\$(PLATFORM)\lib\DYNINST_LIB\libdyninstAPI.lib \
dbghelp.lib

```

Formatted: Indent: First line: 0.5"

Formatted: Font: 10 pt

Formatted: Font: 10 pt

Formatted: Indent: Left: 0", First line: 0.5"

Formatted: Font: 10 pt

#### 5.36.4 Setting Up the Application Program (mutatee)

On most platforms, any additional code that your mutator might need to call in the mutatee (for example files containing instrumentation functions that were too complex to write directly using the API) can be put into a dynamically loaded shared library, which your mutator program can load into the mutatee at runtime using the `loadLibrary` member function of `BPatch_process`.

To locate the runtime library that Dyninst needs to load into your program, an additional environment variable must be set. The variable `DYNINSTAPI_RT_LIB` should be set to the full pathname of the run time instrumentation library, which should be:

NOTE: DYNINST\_LIB should be set to the location where Dyninst library files were installed

Formatted: Indent: Left: 0.5", First line: 0"

`$DYNINST_ROOT/$PLATFORM/lib/$DYNINST_LIB/libdyninstAPI_RT.so` (UNIX)

`%DYNINST_LIBROOT%/i386-unknown-nt4.0/lib/libdyninstAPI_RT.dll` (Windows)

Figure 1 is an example of how you would modify the link command in your Makefile (on one of the UNIX-based platforms) to handle the extra link step required by the current version of the API. If your Makefile contained the link step shown in Figure 1:

Field Code Changed

Field Code Changed

Field Code Changed

Field Code Changed

(a) You would change it to the version shown in Figure 1.

(b) Note that the additions in Figure 1 are shown in bold.

```

OBJECTS = main.o this.o that.o

LIBDIR = $DYNINST_ROOT/lib/$PLATFORM

bubba.pd: ${OBJECTS}
_____ $(CC) $(OBJECTS) \
_____ <insert any additional modules here> \
_____ -lm -lcurses -ltermcap -o bubba.pd

```

*(b) The Link Command Modified to Run Application. Items in **Bold face** show the changes (additions)*

**Figure 1: Changing Your Makefile to Link an Application as a Dyninst mutatee.**

### 5.46.5 Running the Mutator

At this point, you should be ready to run your application program with your mutator. For example, to start the sample program shown in [Appendix A - Complete Examples](#) ~~Appendix A~~:

```
% retee foo <pid>
```

Field Code Changed

### 5.56.6 Optimizing Dyninst Performance

This section describes how to tune Dyninst for optimum performance. During the course of a run, Dyninst will perform several types of analysis on the binary, make safety assumptions about instrumentation that is inserted, and rewrite the binary (perhaps several times). Given some guidance from the user, Dyninst can make assumptions about what work it needs to do and can deliver significant performance improvements.

There are two areas of Dyninst performance users typically care about. First, the time it takes Dyninst to parse and instrument a program. This is typically the time it takes Dyninst to start and analyze a program, and the time it takes to modify the program when putting in instrumentation. Second, many users care about the time instrumentation takes in the modified mutatee. This time is highly dependent on both the amount and type of instrumentation put in, but it is still possible to eliminate some of the Dyninst overhead around the instrumentation.

The following subsections describe techniques for improving the performance of these two areas.

#### 5.5.16.6.1 Optimizing Mutator Performance

CPU time in the Dyninst mutator is usually consumed by either parsing or instrumenting binaries. When a new binary is loaded, Dyninst will analyze the code looking for instrumentation points, global variables, and attempting to identify functions in areas of code that may not have symbols. Upon user request, Dyninst will also parse debug information from the binary, which includes local variable, line, and type information.

All of these items are parsed lazily, that is Dyninst won't try to generate this information until it is requested. Information is parsed on a per-library basis, so a request for information about a specific library function will cause Dyninst to parse information about all functions in that library. Much of the Dyninst parsing performance problems can be removed, or mitigated, by structuring the mutator application so that it only requests information from Dyninst if and when it needs it.

Not all operations require Dyninst to trigger parsing. Some common operations that lead to parsing are:

- Requesting a BPatch\_ ~~pinst~~Point object
- Any operation on a BPatch\_function other than getting its name

*dyninstAPI*

Debugging information is lazily parsed separately from the rest of the binary parsing. Accessing line, type, or local variable information will cause Dyninst to parse the debug information for all three of these.

Another common source of mutator time is spent re-writing the mutatee to add instrumentation. When instrumentation is inserted into a function, Dyninst may need to rewrite some or all of the function to fit the instrumentation in. If multiple pieces of instrumentation are being inserted into a function, Dyninst may need to rewrite that function multiple times.

If the user knows that they will be inserting multiple pieces of instrumentation into one function, they can batch the instrumentation into one bundle, so that the function will only be re-written once, using the `BPatch_process::beginInsertionSet` and `BPatch_process::endInsertionSet` functions (see section [4.44.34.3](#)). Using these functions can result in a significant performance win when inserting instrumentation in many locations.

Field Code Changed

To use the insertion set functions, add a call to `beginInsertionSet` before inserting instrumentation. Dyninst will start buffering up all instrumentation insertions. After the last piece of instrumentation is inserted, call `finalizeInsertionSet`, and all instrumentation will be atomically inserted into the mutatee, with each function being rewritten at most once.

### **5.5.26.6.2 Optimizing Mutatee Performance**

As instrumentation is inserted into a mutatee, it will start to run slower. The slowdown is heavily influenced by three factors: the number of points being instrumented, the instrumentation itself, and the Dyninst overhead around each piece of instrumentation. The Dyninst overhead comes from pieces of protection code (described in more detail below) that do things such as saving/restoring registers around instrumentation, checking for instrumentation recursion, and performing thread safety checks.

The factor by which Dyninst overhead influences mutatee run-time depends on the type of instrumentation being inserted. When inserting instrumentation that runs a memory cache simulator, the Dyninst overhead may be negligible. On the other-hand, when inserting instrumentation that increments a counter, the Dyninst overhead will dominate the time spent in instrumentation. Remember, optimizing the instrumentation being inserted may sometimes be more important than optimizing the Dyninst overhead. Many users have had success writing tools that make use of Dyninst's ability to dynamically remove instrumentation as a performance improvement.

The instrumentation overhead results from safety and correctness checks inserted by Dyninst around instrumentation. Dyninst will automatically attempt to remove as much of this overhead

as possible, however it sometimes must make a conservative decision to leave the overhead in. Given additional, user-provided information Dyninst can make better choices about what safety checks to leave in. An unoptimized post-Dyninst 5.0 instrumentation snippet looks like the following:

<b>Save General Purpose Registers</b>	In order to ensure that instrumentation doesn't corrupt the program, Dyninst saves all live general purpose registers.
<b>Save Floating Point Registers</b>	Dyninst may decide to separately save any floating point registers that may be corrupted by instrumentation.
<b>Generate A Stack Frame</b>	Dyninst builds a stack frame for instrumentation to run under. This provides the illusion to instrumentation that it is running as its own function.
<b>Calculate Thread Index</b>	Calculate an index value that identifies the current thread. This is primarily used as input to the Trampoline Guard.
<b>Test and Set Trampoline Guard</b>	Test to see if we are already recursively executing under instrumentation, and skip the user instrumentation if we are.
<b>Execute User Instrumentation</b>	Execute any <code>BPatch_snippet</code> code.
<b>Unset Trampoline Guard</b>	Marks the this thread as no longer being in instrumentation
<b>Clean Stack Frame</b>	Clean the stack frame that was generated for instrumentation.
<b>Restore Floating Point Registers</b>	Restore the floating point registers to their original state.
<b>Restore General Purpose Registers</b>	Restore the general purpose registers to their original state.

Dyninst will attempt to eliminate as much of its overhead as is possible. The Dyninst user can assist Dyninst by doing the following:

- **Write `BPatch_snippet` code that avoids making function calls.** Dyninst will attempt to perform analysis on the user written instrumentation to determine which general purpose and floating point registers can be saved. It is difficult to analyze function calls that may be nested arbitrarily deep. Dyninst will not analyze any deeper than two levels of function calls before assuming that the instrumentation clobbers all registers and it needs to save everything.
- In addition, not making function calls from instrumentation allows Dyninst to eliminate its tramp guard and thread index calculation. Instrumentation that does not make a function call cannot recursively execute more instrumentation.
- **Call `BPatch::setTrampRecursive(true)` if instrumentation cannot execute recursively.** If instrumentation must make a function call, but will not execute recursively, then enable trampoline recursion. This will cause Dyninst to stop generating a trampoline

**Formatted:** Bulleted + Level: 1 + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

guard and thread index calculation on all future pieces of instrumentation. An example of instrumentation recursion would be instrumenting a call to `write` with instrumentation that calls `printf`—`write` will start calling `printf` `printf` will re-call `write`.

- **Call `BPatch::setSaveFPR(false)` if instrumentation will not clobber floating point registers.** This will cause Dyninst to stop saving floating point registers, which can be a significant win on some platforms.
- **Use simple `BPatch_snippet` objects when possible.** Dyninst will attempt to recognize, peep-hole optimize, and simplify frequently used code snippets when it finds them. For example, on x86 based platforms Dyninst will recognize snippets that do operations like ‘`var = constant`’ or ‘`var++`’ and turn these into optimized assembly instructions that take advantage of CISC machine instructions.
- **Call `BPatch::setInstrStackFrames(false)` before inserting instrumentation that does not need to set up stack frames.** Dyninst allows you to force stack frames to be generated for all instrumentation. This is useful for some applications (e.g., debugging your instrumentation code) but allowing Dyninst to omit stack frames wherever possible will improve performance. This flag is false by default; it should be enabled for as little instrumentation as possible in order to maximize the benefit from optimizing away stack frames.
- **Avoid conditional instrumentation wherever possible.** Conditional logic in your instrumentation makes it more difficult to avoid saving the state of the flags.
- **Avoid unnecessary instrumentation.** Dyninst provides you with all kinds of information that you can use to select only the points of actual interest for instrumentation. Use this information to instrument as selectively as possible. The best way to optimize your instrumentation, ultimately, is to know *a priori* that it was unnecessary and not insert it.



## APPENDIX A - COMPLETE EXAMPLES

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

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

#include <string>

// Example 1: create an instance of class BPatch
BPatch bpatch;

// Example 2: attaching, creating, or opening a file for rewrite
typedef enum {
    create,
    attach,
    open } accessType_t;
BPatch_addressSpace *startInstrumenting(accessType_t accessType,
                                       const char *name,
                                       int pid, // For attach
                                       const char *argv[]) { // For create
    BPatch_addressSpace *handle = NULL;
    switch (accessType) {
    case create:
        handle = bpatch.processCreate(name, argv);
        break;
    case attach:
        handle = bpatch.processAttach(name, pid);
        break;
    case open:
        handle = bpatch.openBinary(name);
        break;
    }
    return handle;
}

// Example 2: find the entry point for "InterestingProcedure"
std::std::vector<BPatch_point *> *findEntryPoint(BPatch_addressSpace *app) {
    std::std::vector<BPatch_function *> functions;
    std::std::vector<BPatch_point *> *points;

    BPatch_image *appImage = app->getImage();
    appImage->findFunction("InterestingProcedure", functions);
    points = functions[0]->findPoint(BPatch_entry);
    return points;
}

// Example 3: create and insert an increment snippet
void createAndInsertSnippet(BPatch_addressSpace *app,
                           std::std::vector<BPatch_point *> *points) {
    BPatch_image *appImage = app->getImage();
    BPatch_variableExpr *intCounter = app->malloc(*(appImage->findType("int")));
    BPatch_arithExpr addOne(BPatch_assign, *intCounter,
```

```

        BPatch_arithExpr(BPatch_plus, *intCounter,
        BPatch_constExpr(1)));

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

// Example 4: finish things up (continue or write out)
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) {
        appProc->continueExecution();
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    }
    if (appBin) {
        appBin->writeFile(newName);
    }
}

// Example 5: binary analysis
int binaryAnalysis(BPatch_addressSpace *app) {
    BPatch_image *appImage = app->getImage();
    int insns_access_memory = 0;

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

    BPatch_flowGraph *fg = funcs[0]->getCFG();
    std::std::set<BPatch_basicBlock *> blocks;
    fg->getAllBasicBlocks(blocks);

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

        std::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++;
            }
        }
        return insns_access_memory;
    }
}

// Example 6: memory instrumentation
void instrumentMemory(BPatch_addressSpace *app) {
    BPatch_image *appImage = app->getImage();

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

```

```

// Get points for each load and store
std::vector<BPatch_function *> funcs;
appImage->findFunction("InterestingProcedure", funcs);
std::vector<BPatch_point *> *points = funcs[0]->findPoint(accessTypes);

// Create a snippet that calls printf with each effective address
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);
std::vector<BPatch_function *> printfFuncs;
appImage->findFunction("printf", printfFuncs);
BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

app->insertSnippet(printfCall, *points);
}

int main() {
    const char *progName = "InterestingProgram"; // = ...
    int progPID = 42; // = ...
    const char *progArgv[] = {"InterestingProgram", "-h", NULL}; // = ...

    // Example 1: create/attach/open a binary
    BPatch_addressSpace *app = startInstrumenting(create, // or attach or open
                                                    progName,
                                                    progPID,
                                                    progArgv);

    // Example 2: get entry point
    std::vector<BPatch_point *> *entryPoint = findEntryPoint(app);

    // Example 3: create and insert increment snippet
    createAndInsertSnippet(app, entryPoint);

    // Example 4: finish up instrumenting
    finishInstrumenting(app, progName);

    // Example 5: get a count of memory accesses
    int insns_access_memory = binaryAnalysis(app);

    // Example 6: instrument memory accesses
    instrumentMemory(app);

    return 0;
}

```

The second example is a program called “re-tee.” It 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. 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 "BPatch.h"
#include "BPatch_point.h"
#include "BPatch_process.h"
#include "BPatch_function.h"
#include "BPatch_Vectorstd::vector.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

```

```

BPatch_Vector<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
BPatch_Vector<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
    BPatch_Vectorstd::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.
    BPatch_Vectorstd::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.

    BPatch_Vectorstd::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
    BPatch_Vectorstd::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
    BPatch_Vectorstd::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();
BPatch_Vectorstd::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;
}

```



## APPENDIX B - RUNNING THE TEST CASES

This section describes how to run the Dyninst test cases. The primary purpose of the test cases is to verify that the API has been installed correctly (and for use in regression testing by the developers of the Dyninst library). The code may also be of use to others since it provides a fairly complete example of how to call most of the API methods. The test suite consists of mutator programs and their associated mutatee programs.

To compile the test suite, type `make` in the appropriate platform specific directory under `dyninst/testsuite`. To run, execute `runTests`. Each test will be executed and the result (PASSED/FAILED/CRASHED) printed.

Test mutators are run by the `test_driver` executable (`test_driver.exe` on Windows). The `test_driver` loads a mutator test from a shared object and runs it on a test mutatee. A single run of the `test_driver` may execute multiple tests (depending on parameters passed), and each test may execute multiple times with different parameters and on different mutatees.

Dyninst's test space can be very large. Each mutatee can be run under different tests, compiled by different compilers, and run with different parameters. For example, one point in this space would be the `test1` mutatee being run under `test1_13`, when compiled with the `g++` compiler, and in `attach` mode. When run without any options, the `test_driver` will run all test combinations that are valid on the current platform. Many of the options that are passed to `test_driver` can be used to limit the test space that it runs in.

In order to prevent a crashing test from stopping the `test_driver` from running subsequent tests, `test_driver` can be run under a wrapper application, `runTests`. The `runTests` wrapper invokes the `test_driver` with the any arguments that were passed to `runTests`. It will watch the `test_driver` process, and if `test_driver` exits with a fault it will print an appropriate error message and restart the `test_driver` on the next test.

It is generally recommended that `runTests` be used when running a large sequence of tests, and `test_driver` be used when debugging issues with a single test.

The `test_driver` and `runTests` applications can be invoked with the following list of arguments. Most arguments are used to limit the space of tests that the testsuite will run. For example, to run the above `test1_13` example, you could use the following command line:

```
test_driver -run test1_13 -mutatee test1.mutatee_g++ -attach
```

`-attach`

Only run tests that attach to the mutatees.

`-create`

Only run tests that create mutatees.

`-rewriter`

Only run tests that rewrite mutatees.

`-staticlink`

Run rewriter tests that use statically linked mutatees.

`-dynamiclink`

Run rewriter tests that use dynamically linked mutatees.

`-allmode`

Run tests for all modes (create, attach, rewriter on statically linked binaries, rewriter on dynamically linked binaries).

`-gcc, -g++, -pgcc, -pgCC, -icc, -icpc`

Run tests on mutatees built with the specified compiler.

`-noclean`

Don't remove rewritten mutatees after running rewriter tests.

`-all`

Run tests for all possible combinations of unoptimized mutatees.

`-none, -low, -high, -max`

Only run tests for mutatees of the given optimization level.

`-allopt`

Run tests for all mutatee optimization levels.

`-full`

Run tests for all possible combinations of mutatees, including all optimization levels. Requires `make all` to build the optimized mutatees.

`-dyninst, -symtab, -instruction, -proccontrol, -stackwalker`

Only run tests for the specified component.

`-allcomp`

Run tests for all components.

`-32, -64`

Only run tests for 32-bit or 64-bit mutatees. This option is only valid on platforms such as AMD64/Linux and PowerPC/Linux where both 32-bit and 64-bit build environments are available.

`-pic`

Only run tests for mutatees compiled as position-independent code. Default is non-PIC mutates. `-all` and `-full` include both.

`-sp, -mp, -st, -mt`

ProcControl-specific: run tests in single process, multiprocess, single thread, multithread modes, respectively.

`-j n`

This option spawns up to `n` `test_driver` instances from a given `runTests` invocation. This option is highly recommended for large test runs.

`-hosts host1 [host2 ... hostn]`

In conjunction with the `-j` option above, will distribute tests over the hosts *host1...hostn*. The hosts must share a filesystem with the machine from which `runTests` is being run, and `ssh` must be configured to allow password-less authentication to those hosts. Note that `-j` controls the total number of `test_driver` instances, not the number per host, so you will need to use a `-j N` at least equal to the number of hosts you wish to use.

`-mutatee <mutatee_name>`

Only run tests that use the specified mutatee name. Only certain mutates can be run with certain tests. The primary test number specifies which mutates it can be run with. For example, all of the `test1_*` tests can be run with the `test1.mutatee_*` mutates, and all of the `test2_*` tests can be run with the `test2.mutatee_*` mutates.

`-run <subtest> <subtest> ...`

Only runs the specific sub-tests listed. For example, to run sub-test case 4 of `test2` you would enter `test_driver -run test2_4`.

`-test`

Alias for `-run`.

`-log`

Print more detailed output, including messages generated by the tests. Without this option the testsuite will capture and hide any messages printed by the test, only showing a summary of whether the test passed or failed. By default, output is sent to `stdout`.

`-logfile <filename>`

Send output from the `-log` option to the given filename rather than to `stdout`.

`-verbose`

Enables test suite debugging output. This is useful when trying to track down issues in the test suite or tests.

Formatted: Heading 1,H1

## APPENDIX 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 exists, the current behavior is that the mutatee is left suspended. To make sure the application continues, call detach with the appropriate 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 semantics are still being specified. Currently, a message is printed to indicate that this has been attempted, 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 periodically call one of the functions `BPatch::pollForStatusChange`, `BPatch::waitForStatusChange`, `BPatch_thread::isStopped`, or `BPatch_thread::isTerminated`.

### 64-bit binaries (AIX)

Dyninst does not support 64-bit binaries on AIX.

### Missing or out-of-date DbgHelp DLL (Windows)

Dyninst requires an up-to-date DbgHelp library on Windows. See the section on Windows-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.

### When Building Dyninst from Source

Commonly, required external libraries and headers (such as libdwarf or libelf) are not found correctly by the compiler. Often, such packages are installed, but outside of the compiler's default search path.

Because of this, we have provided the following extension to our make system. If the file `make.config.local` exists inside the `$DYNINST_ROOT/dyninst` directory, it will automatically be included during the build process.

You can then set the makefile variables `FIRST_INCLUDE` and `FIRST_LIBDIR` inside `make.config.local`. These variables represent the compiler flags used during the compilation and linking phase, respectively. These paths will be searched before any others, insuring that the correct package is used. For example:

```
FIRST_INCLUDE=-I/usr/local/packages/libelf-0.8.5/include
FIRST_LIBDIR =-L/usr/local/packages/libelf-0.8.5/lib
```

# APPENDIX D BUILDING DYNINST

THIS APPENDIX DESCRIBES HOW TO BUILD DYNINST FROM SOURCE CODE, WHICH CAN BE DOWNLOADED FROM [HTTP://WWW.PARADYN.ORG](http://www.paradyne.org) OR [HTTP://WWW.DYNINST.ORG](http://www.dyninst.org).

## BUILDING ON UNIX

BUILDING DYNINST ON UNIX PLATFORMS IS A FOUR STEP PROCESS THAT INVOLVES: UNPACKING THE DYNINST SOURCE, INSTALLING ANY DYNINST DEPENDENCIES, CONFIGURING PATHS IN MAKE.CONFIG.LOCAL, AND RUNNING THE BUILD.

DYNINST'S SOURCE CODE IS PACKAGED IN A TAR.GZ FORMAT. IF YOUR DYNINST SOURCE TARBALL IS CALLED SRCDIST\_V5.0.TAR.GZ, THEN YOU COULD EXTRACT IT WITH THE COMMAND `GUNZIP SRCDIST_V5.0.TAR.GZ & TAR -XVF SRCDIST_V5.0.TAR`. THIS WILL CREATE TWO DIRECTORIES: DYNINST AND SCRIPTS.

DYNINST HAS SEVERAL DEPENDENCIES, DEPENDING ON WHAT PLATFORM YOU ARE USING, WHICH MUST BE INSTALLED BEFORE DYNINST CAN BE BUILT. NOTE THAT FOR MOST OF THESE PACKAGES DYNINST NEEDS TO BE ABLE TO ACCESS THE PACKAGE'S INCLUDE FILES, WHICH MEANS THAT DEVELOPMENT VERSIONS ARE REQUIRED. IF A VERSION NUMBER IS LISTED FOR A PACKAGED, THEN THERE ARE KNOWN BUGS THAT MAY AFFECT DYNINST WITH EARLIER VERSIONS OF THE PACKAGE.

LINUX/X86	LIBDWARF-20100808
	LIBELF
LINUX/X86-64	LIBDWARF-20100808
	LIBELF
AIX/POWER	NO-EXTERNAL-DEPENDENCIES

Formatted: Heading 1,H1

Formatted: Font: Not Bold

Formatted: English (U.S.)

Formatted: English (U.S.)

Formatted: Heading 1,H1

Formatted: English (U.S.)

Formatted: Heading 1,H1

~~LINUX/POWER LIBDWARF-20100808~~

~~LIBELF~~

Formatted: English (U.S.)

Formatted: Heading 1,H1

Formatted: English (U.S.)

Formatted: Heading 1,H1

Formatted: English (U.S.)

Formatted: Heading 1,H1

Formatted: Heading 1,H1

~~AT THE TIME OF THIS WRITING THE LINUX PACKAGES COULD BE FOUND AT:~~

~~• LIBDWARF [HTTP://REALITY.SGIWEB.ORG/DAVEA/DWARF.HTML](http://reality.sgiweb.org/davea/dwarf.html)~~

~~• LIBELF [HTTP://WWW.MR511.DE/SOFTWARE/ENGLISH.HTML](http://www.mr511.de/software/english.html)~~

~~• LIBUNWIND~~

~~[HTTP://WWW.HPL.HP.COM/RESEARCH/LINUX/LIBUNWIND/DOWNLOAD.PHP4](http://www.hpl.hp.com/research/linux/libunwind/download.php4)~~

Formatted: Heading 1,H1, No bullets or numbering

Formatted: Heading 1,H1

~~ONCE THE DEPENDENCIES FOR DYNINST HAVE BEEN INSTALLED, DYNINST MUST BE CONFIGURED TO KNOW WHERE TO FIND THESE PACKAGES. THIS IS DONE THROUGH DYNINST'S DYNINST/MAKE.CONFIG.LOCAL FILE. THIS FILE MUST BE WRITTEN IN GNU MAKEFILE SYNTAX AND MUST SPECIFY DIRECTORY LOCATIONS FOR EACH DEPENDENCY. SPECIFICALLY, LIBDWARFDIR, LIBELFDIR AND TCLTK\_DIR VARIABLES MUST BE SET. LIBDWARFDIR SHOULD BE SET TO THE ABSOLUTE PATH OF LIBDWARF LIBRARY WHERE DWARF.H AND LIBDWARF.H FILES RESIDE. LIBELFDIR SHOULD BE SET TO THE ABSOLUTE PATH WHERE LIBELF.A AND LIBELF.SO FILES ARE LOCATED.~~

~~MAKE.CONFIG.LOCAL CAN ALSO BE GENERATED VIA AUTOCONF. ON MOST SYSTEMS, CONFIGURE WITH NO ARGUMENTS WILL GENERATE A VALID MAKE.CONFIG.LOCAL. FOR MORE DETAILS, CONFIGURE HELP WILL PROVIDE A FULL LIST OF CONFIGURATION OPTIONS.~~

~~THE NEXT THING IS TO SET DYNINST\_ROOT, PLATFORM, AND LD\_LIBRARY\_PATH ENVIRONMENT VARIABLES. DYNINST\_ROOT SHOULD BE SET TO PATH OF THE DIRECTORY THAT CONTAINS DYNINST AND SCRIPTS SUBDIRECTORIES.~~

~~PLATFORM SHOULD BE SET TO ONE OF THE FOLLOWING VALUES DEPENDING UPON WHAT OPERATING SYSTEM YOU ARE RUNNING ON:~~



~~I386-UNKNOWN-LINUX2.4 — LINUX 2.4/2.6 ON AN INTEL X86 PROCESSOR~~

~~RS6000-IBM-AIX5.1 — AIX VERSION 5.1~~

~~X86\_64-UNKNOWN-LINUX2.4 — LINUX 2.4/2.6 ON AN AMD-64 PROCESSOR~~

~~PPC64-LINUX — LINUX 2.6 ON A 64-BIT PPC PROCESSOR~~

~~PPC32-LINUX — LINUX 2.6 ON A 32-BIT PPC PROCESSOR~~

~~THE LD\_LIBRARY\_PATH VARIABLE SHOULD BE SET IN A WAY THAT IT INCLUDES *LIBDWARF* *HOME* *DIRECTORY/LIB* AND *\$(DYNINST\_ROOT)/\$(PLATFORM)/LIB* DIRECTORIES.~~

~~ONCE MAKE\_CONFIG\_LOCAL IS SET YOU ARE READY TO BUILD DYNINST. CHANGE TO THE DYNINST DIRECTORY AND EXECUTE THE COMMAND *make dyninstAPI*. THIS WILL BUILD DYNINST'S MUTATOR LIBRARY, THE DYNINST RUNTIME LIBRARY, AND DYNINST'S TEST SUITE. SUCCESSFULLY BUILT BINARIES WILL BE STORED IN A DIRECTORY NAMED AFTER YOUR PLATFORM AT THE SAME LEVEL AS THE DYNINST DIRECTORY.~~

## BUILDING ON WINDOWS

~~DYNINST FOR WINDOWS IS BUILT WITH MICROSOFT VISUAL STUDIO 2008 PROJECT AND SOLUTION FILES. BUILDING DYNINST FOR WINDOWS IS SIMILAR TO UNIX IN THAT IT IS A FOUR STEP PROCESS: UNPACK THE DYNINSTAPI SOURCE CODE, INSTALL DYNINST'S PACKAGE DEPENDENCIES, CONFIGURE VISUAL STUDIO TO USE THE DEPENDENCIES, AND RUN THE BUILD SYSTEM.~~

~~DYNINST SOURCE CODE IS DISTRIBUTED AS PART OF A TAR.GZ PACKAGE. MOST POPULAR UNZIPPING PROGRAMS ARE CAPABLE OF HANDLING THIS FORMAT. EXTRACTING THE DYNINST TARBALL RESULTS IN TWO DIRECTORIES: DYNINST AND SCRIPTS.~~

~~DYNINST FOR WINDOWS DEPENDS ON MICROSOFT'S DEBUGGING TOOLS FOR WINDOWS, WHICH COULD BE FOUND AT HTTP://WWW.MICROSOFT.COM/WHDC/DEVTOOLS/DEBUGGING/DEFAULT.MSP X AT THE TIME OF THIS WRITING. DOWNLOAD THESE TOOLS AND INSTALL~~

Formatted: Heading 1,H1, Indent: First line: 0"

Formatted: Heading 1,H1

Formatted: Font: Not Bold

~~THEM AT AN APPROPRIATE LOCATION. MAKE SURE TO DO A CUSTOM INSTALL AND INSTALL THE SDK, WHICH IS NOT ALWAYS INSTALLED BY DEFAULT. FOR THE REST OF THIS SECTION, WE WILL ASSUME THAT THE DEBUGGING TOOLS ARE INSTALLED AT C:\PROGRAM FILES\DEBUGGING TOOLS FOR WINDOWS. IF THIS IS NOT THE CASE, THEN ADJUST THE FOLLOWING INSTRUCTION APPROPRIATELY.~~

~~ONCE THE DEBUGGING TOOLS ARE INSTALLED, VISUAL STUDIO MUST BE CONFIGURED TO USE THEM. WE NEED TO ADD THE DEBUGGING TOOLS INCLUDE AND LIBRARY DIRECTORIES TO VISUAL STUDIO'S SEARCH PATHS. IN VISUAL STUDIO 2008 SELECT OPTIONS... FROM THE TOOLS MENU. NEXT SELECT PROJECTS AND VC++ DIRECTORIES FROM THE PANE ON THE LEFT. YOU SHOULD SEE A LIST OF DIRECTORIES THAT ARE SORTED INTO CATEGORIES SUCH AS 'EXECUTABLE FILES', 'INCLUDE FILES', ETC. THE CURRENT CATEGORY CAN BE CHANGED WITH A DROP DOWN BOX IN THE UPPER RIGHT HAND SIDE OF THE DIALOG.~~

~~FIRST, CHANGE TO THE 'LIBRARY FILES' CATEGORY, AND ADD AN ENTRY THAT POINTS TO C:\PROGRAM FILES\DEBUGGING TOOLS FOR WINDOWS\SDK\LIB\I386. MAKE SURE THAT THIS ENTRY IS ABOVE VISUAL STUDIO'S DEFAULT SEARCH PATHS.~~

~~NEXT, CHANGE TO THE 'INCLUDE FILES' CATEGORY AND MAKE A NEW ENTRY IN THE LIST THAT POINTS TO C:\PROGRAM FILES\DEBUGGING TOOLS FOR WINDOWS\SDK\INC. ALSO MAKE SURE THAT THIS ENTRY IS ABOVE VISUAL STUDIO'S DEFAULT SEARCH PATHS. SOME USERS HAVE HAD A PROBLEM WHERE VISUAL STUDIO CANNOT FIND THE CVCONST.H FILE. YOU MAY NEED TO ADD THE DIRECTORY CONTAINING THIS FILE TO THE INCLUDE SEARCH PATH. WE HAVE SEEN IT INSTALLED AT \$(VCINSTALLDIR)/../VISUAL STUDIO SDKS/DIA SDK/INCLUDE, ALTHOUGH YOU MAY NEED TO SEARCH FOR IT.~~

~~ONCE YOU HAVE INSTALLED AND CONFIGURED THE DEBUGGING TOOLS FOR WINDOWS YOU ARE READY TO BUILD DYNINST. FIRST, YOU NEED TO CREATE THE DIRECTORIES WHERE DYNINST WILL INSTALL ITS COMPLETED BUILD. FROM THE DYNINST DIRECTORY YOU NEED TO CREATE THE DIRECTORIES ../I386-UNKNOWN-NT4.0/BIN AND ../I386-UNKNOWN-NT4.0/LIB. NEXT OPEN THE SOLUTION FILE DYNINST/DYNINSTAPI.SLN WITH VISUAL STUDIO. YOU CAN THEN BUILD DYNINST BY SELECT 'BUILD~~

~~SOLUTION' FROM THE BUILD MENU. THIS WILL BUILD THE DYNINST  
MUTATOR LIBRARY, THE RUNTIME LIBRARY, AND THE TEST SUITE.~~

**Formatted:** Font: Not Bold

**A**[attachProcess · 11](#)**B**

[BPatch\\_addrSpec NP · 60](#)  
[BPatch\\_arithExpr · 42](#)  
[BPatch\\_basicBlockLoop · 54](#)  
[BPatch\\_boolExpr · 43](#)  
[BPatch\\_breakPointExpr · 43](#)  
[BPatch\\_bytesAccessedExpr · 10, 43](#)  
[BPatch\\_cblock · 56](#)  
[BPatch\\_constExpr · 43](#)  
[BPatch\\_countSpec NP · 60](#)  
[BPatch\\_effectiveAddressesExpr · 44](#)  
[BPatch\\_flowGraph · 49](#)  
[BPatch\\_funcCallExpr · 44](#)  
[BPatch\\_function · 31](#)  
[BPatch\\_ifExpr · 45](#)  
[BPatch\\_image · 36](#)  
[BPatch\\_memoryAccess · 59](#)  
[BPatch\\_module · 39](#)  
[Bpatch\\_nullExpr · 46](#)  
[BPatch\\_opCode · 33](#)  
[Bpatch\\_paramExpr · 45, 46](#)  
[BPatch\\_point · 34](#)  
[BPatch\\_retExpr · 42, 44, 45, 46](#)  
[BPatch\\_sequence · 46](#)  
[BPatch\\_Set · 57](#)  
[BPatch\\_snippet · 42](#)  
[BPatch\\_sourceBlock · 56](#)  
[BPatch\\_sourceObj · 30](#)  
[BPatch\\_tidExpr · 46](#)  
[BPatch\\_type · 47](#)  
[BPatch\\_variableExpr · 48](#)  
[BPatch\\_Vector · 57](#)  
[BPatchErrorCallback · 17, 19](#)  
[BPatchErrorLevel · 17, 19, 27, 30](#)  
[BPatchPostForkCallback · 18](#)  
[BPatchThreadEventCallback · 18, 19](#)

**C**

[Class BPatch\\_basicBlock · 51, 52](#)  
[continueExecution · 25](#)  
[createArray · 15](#)  
[createEnum · 15](#)  
[createInstPointAtAddr · 36](#)  
[createPointer · 16](#)  
[createProcess · 11](#)  
[createScalar · 15](#)

[createStruct · 16](#)  
[createTypedef · 16](#)  
[createUnion · 16](#)

**D**

[deleteSnippet · 22](#)  
[detach · 25, 26, 27](#)  
[dominates · 53](#)  
[dumpCore · 29](#)

**E**

[findFunction · 37, 39, 57](#)  
[findPoint · 33](#)  
[findType · 38](#)  
[findVariable · 38](#)  
[free · 21](#)  
[funcJumpExpr · 44](#)

**G**

[getAddress · 35](#)  
[getAllBasicBlocks · 49](#)  
[getAllDominates · 53](#)  
[getBaseAddr · 34, 49](#)  
[getBlockNumber · 53](#)  
[getByteCount NP · 60](#)  
[getCalledFunction · 34](#)  
[getCallStack · 28, 29](#)  
[getCblocks · 47](#)  
[getCFG · 34](#)  
[getComponents · 47, 49, 56](#)  
[getConstituentType · 47](#)  
[getContainedLoops · 55](#)  
[getCost · 42](#)  
[getCurrentSnippets · 35](#)  
[getDataClass · 47](#)  
[getEntryBasicBlock · 49](#)  
[getExitBasicBlock · 49](#)  
[getFP · 57](#)  
[getFrameType · 57](#)  
[getFunctions · 56](#)  
[getHigh · 47](#)  
[getImage · 20, 38, 39, 40](#)  
[getImm · 60](#)  
[getImmediateDominates · 53](#)  
[getImmediateDominator · 53](#)  
[getInheritedVariable · 26](#)  
[getLanguage · 31](#)  
[getLoopBasicBlocks · 55](#)  
[getLoopHead · 55](#)  
[getLoops · 49, 50](#)

**Field Code Changed****Formatted:** Number of columns: 2

[getLow](#) · 47  
[getMangledName](#) · 31, 32  
[getMemoryAccess](#) · 35, 53  
[getModule](#) · 33  
[getModuleName](#) · 33  
[getModules](#) · 36  
[getName](#) · 31, 41, 48  
[getObjParent](#) · 31  
[getParams](#) · 32  
[getPC](#) · 57  
[getPointType](#) · 34  
[getProcedures](#) · 36, 40  
[getReg](#) · 60  
[getReturnType](#) · 32  
[getSourceBlock](#) · 53  
[getSourceFile](#) · 56  
[getSourceLines](#) · 56  
[getSourceObj](#) · 30  
[getSources](#) · 50, 52  
[getSrcType](#) · 30  
[getStartAddr](#) NP · 60  
[getTargets](#) · 52  
[getThreads](#) · 11  
[getType](#) · 42  
[getUniqueString](#) · 41

**I**

[insertSnippet](#) · 22  
[isALoad](#) NP · 59  
[isAPrefetch](#) NP · 60  
[isASTore](#) NP · 59  
[isCompatible](#) · 48  
[isInstrumentable](#) · 32  
[isSharedLib](#) · 32, 41  
[isStopped](#) · 26  
[isTerminated](#) · 26

**L**

[libraryName](#) · 32

**M**

[malloc](#) · 21  
[Memory Access Classes](#) · 59  
[Memory Access Snippets](#) · 10

**A**

[attachProcess](#) · 11

**O**

[oneTimeCode](#) · 26, 27, 29

**P**

[pollForStatusChange](#) · 12, 13  
[prefetchType](#) NP · 60

**R**

[readValue](#) · 48  
[registerDynamicLinkCallback](#) · 19  
[registerErrorCallback](#) · 17, 18  
[registerExecCallback](#) · 18  
[registerExitCallback](#) · 19  
[registerPostForkCallback](#) · 18  
[registerPreForkCallback](#) · 18  
[removeFunctionCall](#) · 23  
[replaceFunction](#) · 23  
[replaceFunctionCall](#) · 23

**S**

[setDebugParsing](#) · 12  
[setInheritSnippets](#) · 25  
[setTrampRecursive](#) · 12  
[setTypeChecking](#) · 13  
[stopExecution](#) · 25  
[stopSignal](#) · 26

**T**

[terminateExecution](#) · 25  
[Type Checking](#) · 61

**U**

[usesTrap](#) NP · 35

**W**

[writeValue](#) · 48

**B**

[BPatch\\_addrSpec](#) NP · 60  
[BPatch\\_arithExpr](#) · 42  
[BPatch\\_basicBlockLoop](#) · 54

Formatted: Number of columns: 2

[BPatch\\_boolExpr](#)—42  
[BPatch\\_breakPointExpr](#)—43  
[BPatch\\_bytesAccessedExpr](#)—10, 43  
[BPatch\\_cblock](#)—56  
[BPatch\\_constExpr](#)—43  
[BPatch\\_countSpec](#)—NP—60  
[BPatch\\_effectiveAddressesExpr](#)—42  
[BPatch\\_flowGraph](#)—48  
[BPatch\\_funcCallExpr](#)—44  
[BPatch\\_function](#)—31  
[BPatch\\_HExpr](#)—44  
[BPatch\\_image](#)—25  
[BPatch\\_memoryAccess](#)—50  
[BPatch\\_module](#)—38  
[Bpatch\\_nullExpr](#)—46  
[Bpatch\\_opcode](#)—33  
[Bpatch\\_paramExpr](#)—44, 45  
[Bpatch\\_point](#)—34  
[Bpatch\\_refExpr](#)—42, 43, 45, 46  
[Bpatch\\_sequence](#)—46  
[Bpatch\\_Set](#)—57  
[Bpatch\\_snippet](#)—41  
[Bpatch\\_sourceBlock](#)—56  
[Bpatch\\_sourceObj](#)—30  
[Bpatch\\_tidExpr](#)—46  
[Bpatch\\_type](#)—46  
[Bpatch\\_variableExpr](#)—47  
[Bpatch\\_Vector](#)—57  
[BpatchErrorCallback](#)—17, 18, 19, 20  
[BpatchErrorLevel](#)—17, 19, 26, 30  
[BpatchPostForkCallback](#)—18  
[BpatchThreadEventCallback](#)—18, 19

## C

[Class BPatch\\_basicBlock](#)—50, 52  
[continueExecution](#)—25  
[createArray](#)—15  
[createEnum](#)—15  
[createInstPointAtAddr](#)—35  
[createPointer](#)—16  
[createProcess](#)—11  
[createScalar](#)—15  
[createStruct](#)—16  
[createTypedef](#)—16  
[createUnion](#)—16

## D

[deleteSnippet](#)—22  
[detach](#)—24, 26  
[dominates](#)—52  
[dumpCore](#)—20

## F

[findFunction](#)—36, 37, 39, 57

[findPoint](#)—32  
[findType](#)—38  
[findVariable](#)—37  
[free](#)—21  
[funcJumpExpr](#)—44

## G

[getAddress](#)—34  
[getAllBasicBlocks](#)—49  
[getAllDominates](#)—52  
[getBaseAddr](#)—33, 48  
[getBlockNumber](#)—53  
[getByteCount](#)—NP—60  
[getCalledFunction](#)—34  
[getCallStack](#)—28, 29  
[getCblocks](#)—46  
[getCFG](#)—33, 34  
[getComponents](#)—46, 48, 56  
[getConstituentType](#)—47  
[getContainedLoops](#)—54  
[getCost](#)—42  
[getCurrentSnippets](#)—35  
[getDataClass](#)—47  
[getEntryBasicBlock](#)—49  
[getExitBasicBlock](#)—49  
[getFP](#)—57  
[getFrameType](#)—56  
[getFunctions](#)—56  
[getHigh](#)—47  
[getImage](#)—20, 38, 39, 40  
[getImm](#)—60  
[getImmediateDominates](#)—52  
[getImmediateDominators](#)—52  
[getInheritedVariable](#)—25  
[getLanguage](#)—31  
[getLoopBasicBlocks](#)—54, 55  
[getLoopHead](#)—55  
[getLoops](#)—49  
[getLow](#)—47  
[getMangledName](#)—31, 32  
[getMemoryAccess](#)—25, 52  
[getModule](#)—32  
[getModuleName](#)—32  
[getModules](#)—36  
[getName](#)—31, 40, 47  
[getObjParent](#)—30  
[getParams](#)—32  
[getPC](#)—57  
[getPointType](#)—34  
[getProcedures](#)—36, 40  
[getReg](#)—60  
[getReturnType](#)—32  
[getSourceBlock](#)—52  
[getSourceFile](#)—56  
[getSourceLines](#)—56  
[getSourceObj](#)—30  
[getSources](#)—50, 51, 52  
[getSrcType](#)—30

[getStartAddr NP — 60](#)  
[getTargets — 52](#)  
[getThreads — 11](#)  
[getType — 41](#)  
[getUniqueString — 41](#)

## **I**

[insertSnippet — 22](#)  
[isALoad NP — 59](#)  
[isAPrefetch NP — 59](#)  
[isAStore NP — 59](#)  
[isCompatible — 47](#)  
[isInstrumentable — 32](#)  
[isSharedLib — 32, 41](#)  
[isStopped — 25](#)  
[isTerminated — 25](#)

## **L**

[libraryName — 32](#)

## **M**

[malloc — 21](#)  
[Memory Access Classes — 59](#)  
[Memory Access Snippets — 10](#)

## **O**

[oneTimeCode — 26, 27, 29](#)

## **P**

[pollForStatusChange — 12, 13](#)  
[prefetchType NP — 59](#)

## **A**

[attachProcess — 13](#)

## **B**

[BPatch\\_addrSpec NP — 67](#)  
[BPatch\\_arithExpr — 46](#)  
[BPatch\\_basicBlockLoop — 58](#)  
[BPatch\\_boolExpr — 47](#)  
[BPatch\\_breakPointExpr — 47](#)  
[BPatch\\_bytesAccessedExpr — 11, 47](#)  
[BPatch\\_cblock — 60](#)

## **R**

[readValue — 48](#)  
[registerDynamicLinkCallback — 20](#)  
[registerErrorCallback — 17, 18](#)  
[registerExecCallback — 18](#)  
[registerExitCallback — 19](#)  
[registerPostForkCallback — 19](#)  
[registerPreForkCallback — 19](#)  
[removeFunctionCall — 23](#)  
[replaceFunction — 23](#)  
[replaceFunctionCall — 24](#)

## **S**

[setDebugParsing — 12](#)  
[setInheritSnippets — 23](#)  
[setMutationsActive — 26](#)  
[setTrampRecursive — 12](#)  
[setTypeChecking — 13](#)  
[stopExecution — 25](#)  
[stopSignal — 25](#)

## **T**

[terminateExecution — 25](#)  
[Type Checking — 60, 61](#)

## **U**

[usesTrap NP — 34](#)

## **W**

[writeValue — 48](#)

[BPatch\\_constExpr — 47](#)  
[BPatch\\_countSpec NP — 67](#)  
[BPatch\\_effectiveAddressesExpr — 48](#)  
[BPatch\\_flowGraph — 53](#)  
[BPatch\\_funcCallExpr — 48](#)  
[BPatch\\_function — 33](#)  
[BPatch\\_ifExpr — 49](#)  
[BPatch\\_image — 39](#)  
[BPatch\\_memoryAccess — 66](#)  
[BPatch\\_module — 43](#)  
[BPatch\\_nullExpr — 50](#)  
[BPatch\\_opCode — 35](#)  
[BPatch\\_paramExpr — 49, 50](#)  
[BPatch\\_point — 38](#)  
[BPatch\\_retExpr — 46, 48, 49, 50](#)  
[BPatch\\_sequence — 50](#)  
[BPatch\\_Set — 64](#)

[BPatch\\_snippet](#)—46  
[BPatch\\_sourceBlock](#)—60  
[BPatch\\_sourceObj](#)—32  
[BPatch\\_tidExpr](#)—50  
[BPatch\\_type](#)—51  
[BPatch\\_variableExpr](#)—52  
[BPatch\\_Vector](#)—64  
[BPatchErrorCallback](#)—19, 20, 21, 22  
[BPatchErrorLevel](#)—19, 21, 28, 32  
[BPatchPostForkCallback](#)—20  
[BPatchThreadEventCallback](#)—20, 21

## C

[Class BPatch\\_basicBlock](#)—55, 56  
[continueExecution](#)—27  
[createArray](#)—17  
[createEnum](#)—17  
[createInstPointAtAddr](#)—40  
[createPointer](#)—18  
[createProcess](#)—13  
[createScalar](#)—17  
[createStruct](#)—18  
[createTypedef](#)—18  
[createUnion](#)—18

## D

[deleteSnippet](#)—24  
[detach](#)—26, 28  
[dominates](#)—57  
[dumpCore](#)—31

## F

[findFunction](#)—41, 43, 61  
[findPoint](#)—35  
[findType](#)—42  
[findVariable](#)—42  
[free](#)—23  
[funcJumpExpr](#)—48

## G

[getAddress](#)—38  
[getAllBasicBlocks](#)—53  
[getAllDominates](#)—57  
[getBaseAddr](#)—35, 53  
[getBlockNumber](#)—57  
[getByteCount\\_NP](#)—67  
[getCalledFunction](#)—38  
[getCallStack](#)—29, 30  
[getCblocks](#)—51  
[getCFG](#)—36  
[getComponents](#)—51, 53, 60

[getConstituentType](#)—51  
[getContainedLoops](#)—59  
[getCost](#)—46  
[getCurrentSnippets](#)—39  
[getDataClass](#)—51  
[getEntryBasicBlock](#)—53  
[getExitBasicBlock](#)—53  
[getFP](#)—61  
[getFrameType](#)—61  
[getFunctions](#)—60  
[getHigh](#)—51  
[getImage](#)—22, 42, 43, 44  
[getImm](#)—67  
[getImmediateDominates](#)—57  
[getImmediateDominator](#)—57  
[getInheritedVariable](#)—27  
[getLanguage](#)—33  
[getLoopBasicBlocks](#)—59  
[getLoopHead](#)—59  
[getLoops](#)—53, 54  
[getLow](#)—51  
[getMangledName](#)—33, 34  
[getMemoryAccess](#)—39, 57  
[getModule](#)—34  
[getModuleName](#)—34  
[getModules](#)—40  
[getName](#)—33, 45, 52  
[getObjParent](#)—32  
[getParams](#)—34  
[getPC](#)—61  
[getPointType](#)—38  
[getProcedures](#)—40, 44  
[getReg](#)—67  
[getReturnType](#)—34  
[getSourceBlock](#)—57  
[getSourceFile](#)—60  
[getSourceLines](#)—60  
[getSourceObj](#)—32  
[getSources](#)—54, 56  
[getSrcType](#)—32  
[getStartAddr\\_NP](#)—66  
[getTargets](#)—56  
[getThreads](#)—13  
[getType](#)—46  
[getUniqueString](#)—45

## I

[insertSnippet](#)—24  
[isALoad\\_NP](#)—66  
[isAPrefetch\\_NP](#)—66  
[isAStore\\_NP](#)—66  
[isCompatible](#)—52  
[isInstrumentable](#)—34  
[isSharedLib](#)—34, 45  
[isStopped](#)—27  
[isTerminated](#)—27



***L***

~~libraryName — 34~~

***M***

~~malloc — 23~~  
~~Memory Access Classes — 66~~  
~~Memory Access Snippets — 11~~

***O***

~~oneTimeCode — 27, 29, 30, 31~~

***P***

~~pollForStatusChange — 14, 15~~  
~~prefetchType\_NP — 66~~

***R***

~~readValue — 52~~  
~~registerDynamicLinkCallback — 22~~  
~~registerErrorCallback — 19, 20~~  
~~registerExecCallback — 20~~  
~~registerExitCallback — 21~~  
~~registerPostForkCallback — 21~~

~~registerPreForkCallback — 21~~  
~~removeFunctionCall — 25~~  
~~replaceFunction — 25~~  
~~replaceFunctionCall — 26~~

***S***

~~setDebugParsing — 14~~  
~~setInheritSnippets — 25~~  
~~setMutationsActive — 28~~  
~~setTrampRecursive — 14~~  
~~setTypeChecking — 15~~  
~~stopExecution — 27~~  
~~stopSignal — 27~~

***T***

~~terminateExecution — 27~~  
~~Type Checking — 67, 68~~

***U***

~~usesTrap\_NP — 39~~

***W***

~~writeValue — 52~~

## REFERENCES

1. B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications (to appear)*, 2000.
2. J. K. Hollingsworth and B. P. Miller, "Using Cost to Control Instrumentation Overhead," *Theoretical Computer Science*, **196**(1-2), 1998, pp. 241-258.
3. 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.
4. 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.
5. J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *PLDI*. June 18-21, 1995, La Jolla, CA, ACM, pp. 291-300.

Field Code Changed