

Paradyn Parallel Performance Tools

dyninstAPI Programmer's Guide

13.0 Release
February 2024

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

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web <https://github.com/dyninst/dyninst>



Contents

1	Introduction	5
2	Abstractions	5
3	Analysis API	6
3.1	Components of a Binary	13
3.1.1	BPatch_addressSpace	13
3.1.2	BPatch_basicBlock	18
3.1.3	BPatch_function	20
3.1.4	BPatch_image	24
3.1.5	BPatch_instruction	27
3.1.6	BPatch_module	28
3.1.7	BPatch_object	31
3.1.8	BPatch_sourceBlock	32
3.1.9	BPatch_sourceObj	33
3.2	Control Flow	34
3.2.1	BPatch_basicBlockLoop	34
3.2.2	BPatch_edge	37
3.2.3	BPatch_flowGraph	38
3.2.4	BPatch_loopTreeNode	39
3.3	Extra Features	41
3.3.1	BPatch_cblock	41
4	Instrumentation API	41
4.1	Dynamic Instrumentation	41
4.1.1	BPatch_process	41
4.1.2	BPatch_thread	44
4.2	Static Instrumentation	45
4.2.1	BPatch_binaryEdit	45

4.3	Stack Analysis and Modification	46
4.3.1	StackMod	46
4.3.2	Insert	46
4.3.3	Remove	46
4.3.4	Move	47
4.3.5	Canary	47
4.3.6	Randomize	47
4.3.7	BPatch_frame	47
4.4	Memory Analysis	48
4.4.1	BPatch_addrSpec_NP	49
4.4.2	BPatch_countSpec_NP	49
4.5	Snippets	49
4.5.1	BPatch_snippet	49
4.5.2	BPatch_actualAddressExpr	50
4.5.3	BPatch_arithExpr	50
4.5.4	BPatch_boolExpr	51
4.5.5	BPatch_breakPointExpr	51
4.5.6	BPatch_bytesAccessedExpr	51
4.5.7	BPatch_constExpr	51
4.5.8	BPatch_dynamicTargetExpr	52
4.5.9	BPatch_effectiveAddressExpr	52
4.5.10	BPatch_funcCallExpr	52
4.5.11	BPatch_ifExpr	52
4.5.12	BPatch_nullExpr	53
4.5.13	BPatch_originalAddressExpr	53
4.5.14	BPatch_paramExpr	53
4.5.15	BPatch_registerExpr	53
4.5.16	BPatch_retExpr	54
4.5.17	BPatch_scrambleRegistersExpr	54

4.5.18	BPatch_sequence	54
4.5.19	BPatch_stopThreadExpr	54
4.5.20	BPatch_tidExpr	54
4.5.21	BPatch_variableExpr	55
4.5.22	BPatch_whileExpr	55
4.6	BPatch Type System	55
5	Error Handling	57
5.1	Asynchronous Callbacks	57
5.2	Code Discovery Callbacks	57
5.3	Code Overwrite Callbacks	57
5.4	Dynamic calls	58
5.5	Dynamic libraries	58
5.6	Errors	59
5.7	Exec	59
5.8	Exit	59
5.9	Fork	60
5.10	One Time Code	61
5.11	Signal Handler	61
5.12	Stopped Threads	61
5.13	User-triggered callbacks	62
6	Conversions to other Dyninst toolkits	62
7	Usage	63
7.1	Creating a mutator program	63
7.2	Setting Up the Application Program (mutatee)	63
7.3	Running the Mutator	64
8	Examples	64
8.1	retree	64

8.2	Instrumenting Memory Access	70
8.3	Instrumenting a Function	72
8.4	Binary Analysis	76
9	Optimization	78
9.1	Optimizing Mutator Performance	78
9.2	Optimizing Mutatee Performance	79
10	Common Pitfalls	81
11	References	82

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 key features of this interface are the abilities to:

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

The goal of this API is to keep the interface small and easy to understand. At the same time, it needs to be sufficiently expressive to be useful for a variety of applications. We accomplished this goal by providing a simple set of abstractions and a way to specify which code to insert into the application.¹

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.

¹To 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.

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

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

The code and data represented by an address space is broken up into function and variable abstractions. Functions contain points, which specify locations to insert instrumentation. Functions also contain a control flow graph abstraction, which contains information about basic blocks, edges, loops, and instructions. If the 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 for a complete description of the type system.

Due to language constructs or compiler optimizations, it may be possible for multiple functions to overlap (that is, share part of the same function body) or for a single function to have multiple entry points. In practice, it is impossible to determine the difference between multiple overlapping functions and a single function with multiple entry points. The DyninstAPI uses a model where each function 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 Analysis API

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

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

Returns the list of processes that are currently defined. This list includes processes that were

directly created by calling `processCreate/processAttach`, and indirectly by the UNIX `fork` or the Windows `CreateProcess` system call. It is up to the user to delete this vector when they are done with it.

```
BPatch_process *processAttach(const char *path, int pid, BPatch_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. There are three modes: `BPatch_normalMode`, `BPatch_exploratoryMode`, and `BPatch_defensiveMode`. Normal mode enables the regular static analysis features of Dyninst. Exploratory mode and defensive mode enable additional dynamic features to correctly analyze programs that contain uncommon code patterns, such as malware. Exploratory mode is primarily oriented towards analyzing dynamic control transfers, while defensive mode additionally aims to tackle code obfuscation and self-modifying code. Both of these modes are still experimental and should be used with caution. Defensive mode is only supported on Windows. Defensive mode has been tested on normal binaries (binaries that run correctly under normal mode), as well as some simple, packed executables (self-decrypting or decompressing). More advanced forms of code obfuscation, such as self-modifying code, have not been tested recently. The traditional Dyninst interface may be used for instrumentation of binaries in defensive mode, but in the case of highly obfuscated code, this interface may prove to be ineffective due to the lack of a complete view of control flow at any given point. Therefore, defensive mode also includes a set of callbacks that enables instrumentation to be performed as new code is discovered. Due to the fact that recent efforts have focused on simpler forms of obfuscation, these callbacks have not been tested in detail. The next release of Dyninst will target more advanced uses of defensive mode.

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

This function opens the executable file or library file pointed to by `path` for binary rewriting. If `openDependencies` is true then Dyninst will also open all shared libraries that `path` depends on. Upon success, this function returns a new instance of a `BPatch_binaryEdit` class that represents

the opened file and any dependent shared libraries. This function 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()
```

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

```
void setTrampRecursive (bool state)
```

Turn on or off trampoline recursion. By default, any snippets invoked while another snippet is active will not be executed. This is the safest behavior, since recursively-calling snippets can cause a program to take up all available system resources and die. For example, adding instrumentation code to the start of `printf`, and then calling `printf` from that snippet will result in infinite recursion. This protection operates at the granularity of an instrumentation point. When snippets are first inserted at a point, this flag determines whether code will be created with recursion protection. Changing the flag is not retroactive, and inserting more snippets will not change the recursion protection of the point. Recursion protection increases the overhead of instrumentation points, so if there is no way for the snippets to call themselves, calling this method with the parameter true will result in a performance gain. The default value of this flag is false.

```
bool isTrampRecursive ()
```

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

`void setTypeChecking(bool state)`

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

`bool isTypeChecked()`

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

`bool waitForStatusChange()`

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

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

`void setInstrStackFrames(bool)`

Turn on and off stack frames in instrumentation. When on, Dyninst will create stack frames around instrumentation. A stack frame allows Dyninst or other tools to walk a call stack through instrumentation, but introduces overhead to instrumentation. The default is to not create stack frames.

`bool getInstrStackFrames()`

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

`void setMergeTramp (bool)`

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

`bool isMergeTramp ()`

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

`void setSaveFPR (bool)`

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

`bool isSaveFPRon ()`

This returns the current status of the floating point saves. True means we are saving 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()`

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

```
void setLivenessAnalysis(bool)
```

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

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

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

```
int getNotificationFD()
```

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

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

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

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

```
BPatch_type *createEnum(const char *name, std::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 , are used with enums created using this function.

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

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

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

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

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

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

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

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

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

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

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

3.1 Components of a Binary

DyninstAPI creates representations for the various components of a binary file. These include the instructions and their semantics, basic blocks, functions, and file-level features like data regions and symbol tables.

3.1.1 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 subclasses.

```
BPatch_image *getImage()
```

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

```
bool getSourceLines(unsigned long addr, std::vector<BPatch_statement> &lines)
```

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`, or false otherwise.

```
bool getAddressRanges(const char *fileName, unsigned int lineNo,  
                      std::vector<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 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, std::string name = std::string(""))
```

```
BPatch_variableExpr *malloc(const BPatch_type &type, std::string name = std::string(""))
```

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 allocates `n` bytes of memory from the heap. The second function 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. If a name is specified, Dyninst will assign `var_name` to the variable; otherwise, it will assign an internal name. 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 free(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). Return true if the free succeeded.

```
bool getRegisters(std::vector<BPatch_register> &regs)
```

Returns the registers available to snippet code.

```
BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr, BPatch_point &point,
                                   BPatch_callWhen when,
                                   BPatch_snippetOrder order = BPatch_firstSnippet)
```

```
BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
                                   const std::vector<BPatch_point *> &points,
                                   BPatch_callWhen when,
                                   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.

```
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 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 Complete Examples.

```
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 * 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. 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, or false otherwise.

```
processType getType()
```

This function returns a processType that reflects whether this address space is a **BPatch_process** or a **BPatch_binaryEdit**.

3.1.2 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(std::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(std::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 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(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(std::set<BPatch_basicBlock*>&)  
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.

```
bool getSourceBlocks(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.

```
std::vector<BPatch_point *> findPoint(const std::set<BPatch_opCode> &ops)  
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.

```
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.

3.1.3 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.

`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.

`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.

`std::vector<BPatch_localVar *> *getVars()`

Returns a vector of `BPatch_localVar` objects that contain the local variables in this function. These 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.

`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 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 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
}
```

```
std::vector<BPatch_point *> *findPoint(const std::set<BPatch_opCode> &ops)
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. Any combination of opcode type 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.

```
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.

```
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.

```
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.

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

Determine which functions overlap with the current function. 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.

```
bool addMods(std::set<StackMod *> mods)
```

Implemented on x86 and x86-64

Apply stack modifications in `mods` to the current function; the `StackMod` class is described in section. Perform error checking, handle stack alignment requirements, and generate any modifications required for cleanup at function exit. `addMods` atomically adds all modifications in `mods`; if any mod is found to be unsafe, none of the modifications in `mods` will be applied.

`addMods` can only be used in binary rewriting mode.

Returns false if the stack modifications are unsafe or if Dyninst is unable to perform the analysis required to guarantee safety.

3.1.4 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`.

```
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.

```
std::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 filename may include path information.

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

Fill sources with the source objects that belong to this image. If there are no source objects, the function returns false. Otherwise, it returns true.

```
std::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.

```
std::vector<BPatch_module *> *getModules()
```

Return a vector of the modules in the image.

```
bool getVariables(std::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.

```
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_function`s 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_function`s 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 (*BPatchFunctionNameSieve) (
    const char *name,
    void* sieve_data
)
```

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

Return a vector of `BPatch_function`s according to the generalized userspecified 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.

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, std::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.

```
BPatch_variableExpr *findVariable(const char *name, bool showError = true)
BPatch_variableExpr *findVariable(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.

```
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<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`.

```
bool getAddressRanges(const char *fileName, unsigned int lineNo,
                      std::vector<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`.

```
bool parseNewFunctions(std::vector<BPatch_module*> &newModules,
                      const std::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, or false otherwise.

3.1.5 `BPatch_instruction`

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()`

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

`bool isASTore()`

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

`bool isAPrefetch_NP()`

Return 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.

3.1.6 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 either 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`.

```
std::vector<BPatch_function*> *findFunction(const char *name,
                                           std::vector<BPatch_function*> &funcs,
                                           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. 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_Vector<BPatch_function *> *findFunctionByAddress(void *addr,
                                                         bool notify_on_failure = true,
                                                         bool incUninstrumentable = false)
```

Return a vector of `BPatch_functions` that contains `addr`, or `NULL` if the function does not exist. If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

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

Returns the function that begins at the specified address `addr`.

```
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.

```
bool getAddressRanges(char *fileName, unsigned int lineNo,
                      std::vector<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::getAddressRange` 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.

`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.

`void *getBaseAddr()`

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

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

Return a vector containing the functions in the module.

`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.

`BPatch_hybridMode getHybridMode()`

Return the mutator's analysis mode for the mutatee; the default mode is the normal mode.

`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.

`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 getSourceLines(unsigned long addr, std::vector<BPatch_statement> &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::getSourceLines` 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`, or false otherwise.

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

Performs a lookup and returns a unique string for this image. Returns a string the 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.

```
bool getVariables(std::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.

```
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).

```
bool isExploratoryModeOn()
```

This function returns true if the mutator's analysis mode sets to the defensive mode or the exploratory mode.

```
bool isMutatee()
```

This function returns true if the module is the mutatee.

```
bool isSharedLib()
```

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

3.1.7 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()
```

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

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

Convert 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;  
}
```

A contiguous sequence of addresses that represents either code or data.

```
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.

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

Return 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.

```
std::vector<BPatch_function*> *  
findFunction(const char *name, std::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. 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 `BPatch_object`, 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.]

3.1.8 `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(std::vector<unsigned short>& srcs)
```

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

3.1.9 BPatch_sourceObj

BPatch_sourceObj is the base class for BPatch_function, BPatch_module, and BPatch_image. 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
}
```

The type of error encountered in a DyninstAPI operation.

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

The type of source statement.

```
BPatch_sourceType getSrcType()
```

Returns the type of the current source object.

```
void getSourceObj(std::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.

```

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
}

```

The programming language of the source.

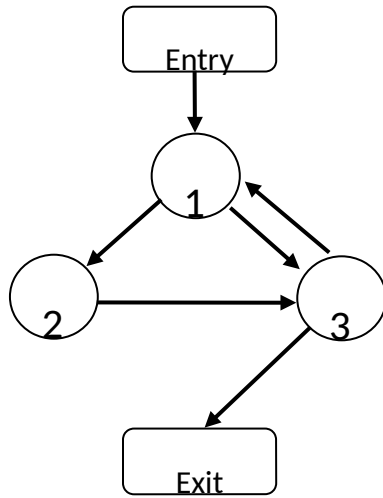
```
BPatch_language getLanguage()
```

Return the source language of the current `BPatch_sourceObject`. For programs that are written in more than one `BPatch_unknownLanguage` will be returned.

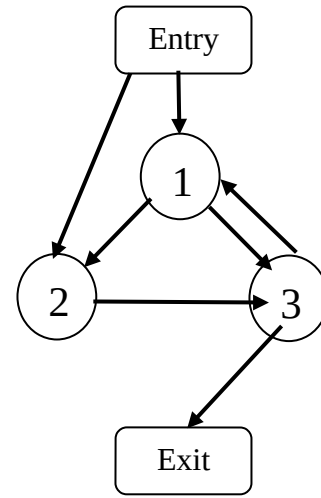
3.2 Control Flow

3.2.1 BPatch_basicBlockLoop

An object of this class represents a loop in the code of the application being instrumented. We detect both natural loops (single-entry loops) and irreducible loops (multi-entry loops). For a natural loop, it has only one entry block and this entry block dominates all blocks in the loop; thus the entry block is also called the head or the header of the loop. However, for an irreducible loop, it has multiple entry blocks and none of them dominates all blocks in the loop; thus there is no head or header for an irreducible loop. The following figure illustrates the difference:



(a) An example of a natural loop



(b) An example of an irreducible loop

Figure (a) above shows a natural loop, where block 1 represents the single entry and block 1 is the head of the loop. Block 1 dominates block 2 and block 3. Figure (b) above shows an irreducible loop, where block 1 and block 2 are the entries of the loop. Neither block 1 nor block 2 dominates block 3.

```
bool containsAddress(unsigned long addr)
```

Return 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)
```

Return 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.

```
int getBackEdges(std::vector<BPatch_edge *> &edges)
```

Returns the number of back edges in this loop and adds those edges to the edges vector. An edge is a back edge if it is from a block in the loop to an entry block of the loop.

```
int getLoopEntries(std::vector<BPatch_basicBlock *> &entries)
```

Returns the number of entry blocks of this loop and adds those blocks to the entries vector. An irreducible loop can have multiple entry blocks.

```
bool getContainedLoops(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.

```
bool getOuterLoops(std::vector<BPatch_basicBlockLoop*>&)
```

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

```
bool getLoopBasicBlocks(std::vector<BPatch_basicBlock*>&)
```

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

```
bool getLoopBasicBlocksExclusive(std::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.

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

Return 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)
```

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

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

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

3.2.2 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.

3.2.3 BPatch_flowGraph

BPatch_flowGraph 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()
```

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

```
void getAllBasicBlocks(std::set<BPatch_basicBlock*>&)  
void getAllBasicBlocks(BPatch_Set<BPatch_basicBlock*>&)
```

Fill the given set with pointers to all basic blocks in the control flow graph.

```
void getEntryBasicBlock(std::vector<BPatch_basicBlock*>&)
```

Fill the given vector with pointers to all basic blocks that are entry points to the function.

```
void getExitBasicBlock(std::vector<BPatch_basicBlock*>&)
```

Fill the given vector with pointers to all basic blocks that are exit points of the function.

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

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

```
void getOuterLoops(std::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  
}
```

The place in the loop structure of the current procedure.

```
std::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.

```
BPatch_basicBlock* findBlockByAddr(Dyninst::Address addr);
```

Find the basic block within this flow graph that contains `addr`. Returns `NULL` on failure. This method is inefficient but guaranteed to succeed if `addr` is present in any block in this CFG. [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.]

3.2.4 BPatch_loopTreeNode

The `BPatch_loopTreeNode` class provides a tree interface to a collection of instances of `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.

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


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 `.y`, where `y` is 1 to `m`, where `m` is the number of sub-loops under the outer loop.

The `foo` function will have a root `BPatch_loopTreeNode`, containing a `NULL` loop entry and two `BPatch_loopTreeNode` children representing the functions 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`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.

3.3 Extra Features

3.3.1 BPatch_cblock

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

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

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

```
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 Instrumentation API

4.1 Dynamic Instrumentation

4.1.1 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.

`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(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_variableExprs` 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 `BPatch_process` of the child 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 `parentSnippet` 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 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.

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

The mechanism by which the process exited.

`BPatch_exitType terminationStatus()`

If the process has exited, `terminationStatus` will indicate whether the process exited normally or because of a signal.

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. Prior to Dyninst 8.2, `getExitCode` would return the argument passed to `exit` or the value returned by `main`; in Dyninst 8.2 and later, it returns the actual exit code as provided by the debug interface and seen by the parent process. In particular, on Linux, this means that exit codes are normalized to the range 0-255.

`int getExitSignal()`

If the process exited because of a received signal, `getExitSignal` will return the associated signal number.

`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.

4.1.2 BPatch_thread

The `BPatch_thread` class represents and controls a thread of execution that is running in a process.

```
void getCallStack(std::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`.

```
dynthread_t 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 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 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 *getInitialFunc()
```

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)
```

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 or 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.

4.2 Static Instrumentation

4.2.1 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, or false otherwise.

4.3 Stack Analysis and Modification

4.3.1 StackMod

Stack modifications are based on the abstraction of stack locations, not the contents of these locations. All stack offsets are with respect to the original stack frame, even if `BPatch_function::addMods` is called multiple times for a single function. Only implemented on x86 and x86-64.

This class defines modifications to the stack frame layout of a function.

4.3.2 Insert

```
Insert(int low, int high)
```

This constructor creates a stack modification that inserts stack space in the range `[low, high)`, where `low` and `high` are stack offsets. `BPatch_function::addMods` will find this modification unsafe if any instructions in the function access memory that will be non-contiguous after `[low,high)` is inserted.

4.3.3 Remove

```
Remove(int low, int high)
```

This constructor creates a stack modification that removes stack space in the range `[low, high)`, where `low` and `high` are stack offsets. `BPatch_function::addMods` will find this modification unsafe if any instructions in the function access stack memory in `[low,high)`.

4.3.4 Move

```
Move(int sLow, int sHigh, int dLow)
```

This constructor creates a stack modification that moves stack space [sLow, sHigh) to [dLow, dLow+(sHigh-sLow)). `BPatch_function::addMods` will find this modification unsafe if `Insert(dLow, dLow+(sHigh-sLow))` or `Remove(sLow, sHigh)` are unsafe.

4.3.5 Canary

```
Canary()  
Canary(BPatch_function* failFunc)
```

implemented on Linux, GCC only

This constructor creates a stack modification that inserts a stack canary at function entry and a corresponding canary check at function exit(s). This uses the same canary as GCC's `-fstack-protector`. If the canary check at function exit fails, `failFunc` is called. `failFunc` must be non-returning and take no arguments. If no `failFunc` is provided, `__stack_chk_fail` from `libc` is called; `libc` must be open in the corresponding `BPatch_addressSpace`. This modification will have no effect on functions in which the entry and exit point(s) are the same. `BPatch_function::addMods` will find this modification unsafe if another `Canary` has already been added to the function. Note, however, that this modification can be applied to code compiled with `-fstack-protector`.

4.3.6 Randomize

```
Randomize()  
Randomize(int seed)
```

This constructor creates a stack modification that rearranges the stack-stored local variables of a function. This modification requires symbol information (e.g., DWARF), and only local variables specified by the symbols will be randomized. If `DyninstAPI` finds a stack access that is not consistent with a symbol-specified local, that local will not be randomized. Contiguous ranges of local variables are randomized; if there are two or more contiguous ranges of locals within the stack frame, each is randomized separately. More than one local variable is required for randomization. `BPatch_function::addMods` will return false if `Randomize` is added to a function without local variable information, without local variables on the stack, or with only a single local variable. `srand` is used to generate a new ordering of local variables; if `seed` is provided, this value is provided to `srand` as its seed. `BPatch_function::addMods` will find this modification unsafe if any other modifications have been applied.

4.3.7 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.


```
enum BPatch_frameType {
    BPatch_frameNormal,    // Normal stack frame
    BPatch_frameSignal,    // Signal invocation
    BPatch_frameTrampoline, // Call into instrumentation code
}
```

```
BPatch_frameType getFrameType()
```

Return the type of the stack frame.

```
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.

```
BPatch_thread *getThread()
```

Returns the thread associated with the stack frame.

```
BPatch_point *getPoint()
```

```
BPatch_point *findPoint()
```

For stack frames corresponding to inserted instrumentation, returns the instrumentation point where that instrumentation.

```
bool isSynthesized()
```

Returns true if this frame was artificially created, false otherwise.

4.4 Memory Analysis

Instrumentation points get memory access information attached to them. This information is used by the memory access snippets, but is also available to the API user.

4.4.1 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 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.

4.4.2 BPatch_countSpec_NP

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

4.5 Snippets

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.

4.5.1 BPatch_snippet

```
BPatch_type *getType()
```

Return the type of the snippet. The `BPatch_type` system is described in Type System.

4.5.2 BPatch_actualAddressExpr

`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.

4.5.3 BPatch_arithExpr

```
enum BPatch_binOp {
    BPatch_assign, // assign the value of rOperand to lOperand
    BPatch_plus,   // add lOperand and rOperand
    BPatch_minus,  // subtract rOperand from lOperand
    BPatch_divide, // divide rOperand by lOperand
    BPatch_times,  // multiply rOperand by lOperand
    BPatch_ref,    // Array reference of the form lOperand[rOperand]
    BPatch_seq,    // Define a sequence of two expressions (similar to comma in C)
}
```

The type of binary operation to perform.

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

Perform the requested binary operation.

```
enum BPatch_unOp {
    BPatch_negate, // Returns the negation of an integer
    BPatch_addr,   // Returns a pointer to a BPatch_variableExpr
    BPatch_deref,  // Dereferences a pointer
}
```

The type of unary operation to perform.

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

Perform the requested unary operation.

4.5.4 BPatch_boolExpr

```
enum BPatch_relOp {
    BPatch_lt, // Return lOperand < rOperand
    BPatch_eq, // Return lOperand == rOperand
    BPatch_gt, // Return lOperand > rOperand
    BPatch_le, // Return lOperand <= rOperand
    BPatch_ne, // Return lOperand != rOperand
    BPatch_ge, // Return lOperand >= rOperand
    BPatch_and, // Return lOperand && rOperand (Boolean and)
    BPatch_or, // Return lOperand || rOperand (Boolean or)
}
```

The boolean operation to perform.

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

Define a relational snippet.

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

4.5.5 BPatch_breakPointExpr

```
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`.

4.5.6 BPatch_bytesAccessedExpr

```
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.

4.5.7 BPatch_constExpr

```
BPatch_constExpr(signed int value)
BPatch_constExpr(unsigned int value)
BPatch_constExpr(signed long value)
```

```

BPatch_constExpr(unsigned long value)
BPatch_constExpr(const char *value)
BPatch_constExpr(const void *value)
BPatch_constExpr(long long 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.

4.5.8 `BPatch_dynamicTargetExpr`

```
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.

4.5.9 `BPatch_effectiveAddressExpr`

```
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.

4.5.10 `BPatch_funcCallExpr`

```

BPatch_funcCallExpr(const BPatch_function& func,
                    const std::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.

AMD64 and IA-32 platforms can have any number of arguments. PowerPC can have at most 8.

4.5.11 `BPatch_ifExpr`

```

BPatch_ifExpr(const BPatch_boolExpr &conditional,
              const BPatch_snippet &tClause,

```

```

        const BPatch_snippet &fClause)

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.

4.5.12 BPatch_nullExpr

```
BPatch_nullExpr()
```

Creates a null snippet that can be used as a placeholder. This snippet contains no executable statements.

4.5.13 BPatch_originalAddressExpr

```
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`.

4.5.14 BPatch_paramExpr

```
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`.

4.5.15 BPatch_registerExpr

```

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.

4.5.16 BPatch_retExpr

`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`.

4.5.17 BPatch_scrambleRegistersExpr

`BPatch_scrambleRegistersExpr()`

This snippet sets all General Purpose Registers to the flag value.

4.5.18 BPatch_sequence

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

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

4.5.19 BPatch_stopThreadExpr

`BPatch_stopThreadExpr(const BPatchStopThreadCallback &cb,
const BPatch_snippet &calculation,
bool useCache = false,
BPatch_stInterpret interp = BPatch_noInterp)`

This snippet stops the thread that executes it. It evaluates a calculation snippet and triggers a callback to the user program with the result of the calculation and a pointer to the `BPatch_point` at which the snippet was inserted.

4.5.20 BPatch_tidExpr

`BPatch_tidExpr(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.

4.5.21 BPatch_variableExpr

```
BPatch_variableExpr(char *in_name,  
                    BPatch_addressSpace *in_addSpace,  
                    AddressSpace *as,  
                    AstNodePtr ast_wrapper_,  
                    BPatch_type *type,  
                    void* in_address)  
  
BPatch_variableExpr(BPatch_addressSpace *in_addSpace,  
                    AddressSpace *as,  
                    void *in_address,  
                    int in_register,  
                    BPatch_type *type,  
                    BPatch_storagestorage = BPatch_storageAddr,  
                    BPatch_point *scp = NULL)  
  
BPatch_variableExpr(BPatch_addressSpace *in_addSpace,  
                    AddressSpace *as,  
                    BPatch_localVar *lv,  
                    BPatch_type *type,  
                    BPatch_point *scp)  
  
BPatch_variableExpr(BPatch_addressSpace *in_addSpace,  
                    AddressSpace *ll_addSpace,  
                    int_variable *iv,  
                    BPatch_type *type)
```

Define a variable snippet of the appropriate type. The first constructor is used to get function pointers; the second is used to get forked copies of variable expression, used by malloc; the third is used for local variables; and the last is used by `BPatch_addressSpace::findOrCreateVariable()`.

4.5.22 BPatch_whileExpr

```
BPatch_whileExpr(const BPatch_snippet &condition, const BPatch_snippet &body)
```

This constructor creates a while statement. The first argument, condition, should be a Boolean expression that will be evaluated to decide whether body should be executed. The second argument, body, is the snippet to execute if the condition evaluates to true.

4.6 BPatch 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**.

5 Error Handling

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()`.

5.1 Asynchronous Callbacks

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

bool registerThreadEventCallback(BPatch_asyncEventType type,
                                BPatchAsyncThreadEventCallback cb)

bool removeThreadEventCallback(BPatch_asyncEventType type,
                                BPatchAsyncThreadEventCallback cb)
```

The type parameter can be either one of `BPatch_threadCreateEvent` or `BPatch_threadDestroyEvent`. Different callbacks can be registered for different values of type.

5.2 Code Discovery Callbacks

```
typedef void (*BPatchCodeDiscoveryCallback)(
    BPatch_Vector<BPatch_function*> &newFuncs,
    BPatch_Vector<BPatch_function*> &modFuncs
)

bool registerCodeDiscoveryCallback(BPatchCodeDiscoveryCallback cb)
bool removeCodeDiscoveryCallback(BPatchCodeDiscoveryCallback cb)
```

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

5.3 Code Overwrite Callbacks

```
typedef void (*BPatchCodeOverwriteBeginCallback)(
    BPatch_Vector<BPatch_basicBlock*> &overwriteLoopBlocks
)
```

```

typedef void (*BPatchCodeOverwriteEndCallback)(
    BPatch_Vector<std::pair<Dyninst::Address,int>> &deadBlocks,
    BPatch_Vector<BPatch_function*> &owFuncs,
    BPatch_Vector<BPatch_function*> &modFuncs,
    BPatch_Vector<BPatch_function*> &newFuncs
)

bool registerCodeOverwriteCallbacks(BPatchCodeOverwriteBeginCallback cbBegin,
                                   BPatchCodeOverwriteEndCallback cbEnd)

```

Register a callback at the beginning and end of overwrite events. Only invoke if Dyninst's hybrid analysis mode is set to `BPatch_defensiveMode`.

The `BPatchCodeOverwriteBeginCallback` callback allows the user to remove any instrumentation when 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.

The `BPatchCodeOverwriteEndCallback` callback delivers the effects of the overwrite loop when it is done executing. In many cases no code will have changed.

5.4 Dynamic calls

```

typedef void (*BPatchDynamicCallSiteCallback)(
    BPatch_point*at_point,
    BPatch_function *called_function
)

bool registerDynamicCallCallback(BPatchDynamicCallSiteCallback cb)
bool removeDynamicCallCallback(BPatchDynamicCallSiteCallback cb)

```

The `registerDynamicCallCallback` interface will not automatically instrument any dynamic call site. To make sure the call back function is called, the user needs to explicitly instrument dynamic call sites. One way to achieve this goal is to first get instrumentation points representing dynamic call sites and then call `BPatch_point::monitorCalls` with a NULL input parameter.

5.5 Dynamic libraries

```

typedef void (*BPatchDynLibraryCallback)(
    BPatch_thread *thr,
    BPatch_object *obj,
    bool loaded
)

BPatchDynLibraryCallback registerDynLibraryCallback(BPatchDynLibraryCallback func)

```

Note that in versions previous to 9.1, BPatchDynLibraryCallbacks signature took a BPatch_module instead of a BPatch_object.

5.6 Errors

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

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

```
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.

5.7 Exec

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

```
BPatchExecCallback registerExecCallback(BPatchExecCallback func)
```

Not implemented on Windows.

5.8 Exit

```
enum BPatch_exitType {
```

```

    NoExit,
    ExitedNormally,
    ExitedViaSignal
}

```

Mechanism by which the process exited.

```

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

```

BPatchExitCallback 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. 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.

5.9 Fork

```

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

BPatchForkCallback registerPostForkCallback(BPatchForkCallback 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.

5.10 One Time Code

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

```
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.

5.11 Signal Handler

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

```
bool registerSignalHandlerCallback(BPatchSignalHandlerCallback cb, std::set<long> &signal_numbers)
bool registerSignalHandlerCallback(BPatchSignalHandlerCallback cb, BPatch_Set<long> *signal_numbers)
```

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.

```
bool removeSignalHandlerCallback(BPatchSignalHandlerCallback cb)
```

Removes a callback registered with `registerSignalHandlerCallback`.

5.12 Stopped Threads

```
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). 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.

5.13 User-triggered callbacks

```
typedef void (*BPatchUserEventCallback)(
    BPatch_process *proc,
    void *buf,
    unsigned int bufsize
)
```

```
bool registerUserEventCallback(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.

```
bool removeUserEventCallback(BPatchUserEventCallback cb)
```

Removes a callback registered with `registerUserEventCallback`.

6 Conversions to other Dyninst toolkits

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 *);

```

7 Usage

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.

7.1 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 process). 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.

The following `CMakeLists.txt` shows how create your mutator program with Dyninst.

```

cmake_minimum_required(VERSION 3.15)
project(retee LANGUAGES CXX)

find_package(Dyninst REQUIRED)

add_executable(retee retee.cpp)
target_link_libraries(retee PUBLIC Dyninst::dyninstAPI)

```

7.2 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

```
| \$(DYNINST_LIB)/libdyninstAPI_RT.so (UNIX)  
| \%(DYNINST_LIB)/libdyninstAPI_RT.dll (Windows)
```

7.3 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:

```
retee foo <pid>
```

8 Examples

8.1 retee

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 "BPatch.h"  
#include "BPatch_function.h"  
#include "BPatch_point.h"  
#include "BPatch_process.h"  
#include "BPatch_thread.h"  
#include <fcntl.h>  
#include <stdio.h>  
#include <vector>  
  
/*  
 * 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"
        " note: <filename> is relative to the application process.\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{\textbar}O_CREAT, 0666);

    // (1) Find the open function
    std::vector<BPatch_function *> openFuncs;
    appImage->findFunction("open", openFuncs);

    if (openFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for open()\n");
        return -1;
    }
}

```

```

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

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

// (4) Create two more constant expressions _WRONLY{\textbar}O_CREAT and 0666
BPatch_constExpr fileFlagsExpr(O_WRONLY{\textbar} O_CREAT);
BPatch_constExpr fileModeExpr(0666);

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

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

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

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

// We have used a oneTimeCode to open the file descriptor. However,
// this returns the file descriptor to the mutator - the mutatee has
// no idea what the descriptor is. We need to allocate a variable in
// the mutatee to hold this value for future use and copy the
// (mutator-side) value into the mutatee variable.
// Note: there are alternatives to this technique. We could have
// allocated the variable before the oneTimeCode and augmented the
// snippet to do the assignment. We could also write the file
// descriptor as a constant into any inserted instrumentation.
BPatch_variableExpr *writeFileDescIntoMutatee(BPatch_process *app,
                                                BPatch_image *appImage,
                                                int fileDescriptor) {

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

// (2) Write the value into the variable
// Like memcpy, writeValue takes a pointer
// The third parameter is for functionality called "saveTheWorld",
// which we don't worry about here (and so is false)
bool ret = fdVar->writeValue((void *)&fileDescriptor, sizeof(int), false);
if (ret == false)

```

```

    return NULL;
    return fdVar;
}

// We now have an open file descriptor in the mutatee. We want to
// instrument write to intercept and copy the output. That happens
// here.
bool interceptAndCloneWrite(BPatch_process *app, BPatch_image *appImage,
                             BPatch_variableExpr *fdVar) {
    // (1) Locate the write call
    std::vector<BPatch_function*> writeFuncs;
    appImage->findFunction("write", writeFuncs);
    if (writeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for write()\n");
        return false;
    }

    // (2) Build the call to (our) write. Arguments are:
    // ours: fdVar (file descriptor)
    // parameter: buffer
    // parameter: buffer size
    // Declare a vector to hold these.
    std::vector<BPatch_snippet*> writeArgs = {
        // The file descriptor
        writeArgs.push_back(fdVar)
    };

    // Well, we need the buffer... but that's a parameter to the
    // function we're implementing. That's not a problem - we can grab
    // it out with a BPatch_paramExpr.
    BPatch_paramExpr buffer(1); // Second (0, 1, 2) argument
    BPatch_paramExpr bufferSize(2);
    writeArgs.push_back(&buffer);
    writeArgs.push_back(&bufferSize);

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

    // (3) Identify the BPatch_point for the entry of write. We're
    // instrumenting the function with itself; normally the findPoint
    // call would operate off a different function than the snippet.
    std::vector<BPatch_point*> *points;
    points = writeFuncs[0]->findPoint(BPatch_entry);
    if ((*points).size() == 0) {
        return false;
    }

    // (4) Insert the snippet at the start of write
    return app->insertSnippet(writeCall, *points);
}

```

```

// Note: we have just instrumented write() with a call to
// write(). This would ordinarily be a _bad thing_, as there is
// nothing to stop infinite recursion - write -> instrumentation
// -> write -> instrumentation....
// However, Dyninst uses a feature called a "tramp guard" to
// prevent this, and it's on by default.
// This function is called as an exit callback (that is, called
// immediately before the process exits when we can still affect it)
// and thus must match the exit callback signature:
//
// typedef void (*BPatchExitCallback) (BPatch_thread *, BPatch_exitType)
//
// Note that the callback gives us a thread, and we want a process - but
// each thread has an up pointer.
void closeFile(BPatch_thread *thread, BPatch_exitType) {
    fprintf(stderr, "Exit callback called for process...\n");
    // (1) Get the BPatch_process and BPatch_images
    BPatch_process *app = thread->getProcess();
    BPatch_image *appImage = app->getImage();
    // The code to be generated is:
    // close(fd);
    // (2) Find close
    std::vector<BPatch_function *> closeFuncs;
    appImage->findFunction("close", closeFuncs);
    if (closeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for close()\n");
        return;
    }
    // (3) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> closeArgs;

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

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

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

    // (7) Tell the app to continue to finish it off.
    app->continueExecution();
}

BPatch bpatch;
// In main we perform the following operations.
// 1) Attach to the process and get BPatch_process and BPatch_image
// handles

```

```

// 2) Open a file descriptor
// 3) Instrument write
// 4) Continue the process and wait for it to terminate
int main(int argc, char *argv[]) {
    int pid;
    if (argc != 3) {
        usage();
        exit(1);
    }
    pid = atoi(argv[1]);

    // Attach to the program - we can attach with just a pid; the
    // program name is no longer necessary
    fprintf(stderr, "Attaching to process %d...\n", pid);
    BPatch_process *app = bpatch.processAttach(NULL, pid);
    if (!app)
        return -1;

    // Read the program's image and get an associated image object
    BPatch_image *appImage = app->getImage();
    std::vector<BPatch_function *> writeFuncs;
    fprintf(stderr, "Opening file %s for write...\n", argv[2]);
    int fileDescriptor = openFileForWrite(app, appImage, argv[2]);
    if (fileDescriptor == -1) {
        fprintf(stderr, "ERROR: opening file %s for write failed\n", argv[2]);
        exit(1);
    }

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

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

    fprintf(stderr, "Instrumenting write...\n");
    bool ret = interceptAndCloneWrite(app, appImage, fdVar);
    if (!ret) {
        fprintf(stderr, "ERROR: failed to instrument mutatee\n");
        exit(1);
    }

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

    // Continue the execution...

```

```

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

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

    printf("Done.\n");
    return 0;
}

```

8.2 Instrumenting Memory Access

```

#include <stdio.h>
#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

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

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

    BPatch_addressSpace *handle = NULL;
    switch (accessType) {
    case create:
        handle = bpatch.processCreate(name, argv);
        if (!handle) {
            fprintf(stderr, "processCreate failed\n");
        }
        break;
    case attach:
        handle = bpatch.processAttach(name, pid);
        if (!handle) {
            fprintf(stderr, "processAttach failed\n");
        }
        break;
    case open:
        // Open the binary file; do not open dependencies

```

```

        handle = bpatch.openBinary(name, false);
        if (!handle) {
            fprintf(stderr, "openBinary failed\n");
        }
        break;
    }
    return handle;
}

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

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

    // Scan the function InterestingProcedure
    // and create instrumentation points
    std::vector<BPatch_function *> functions;
    appImage->findFunction(InterestingProcedure, functions);

    std::vector<BPatch_point *> *points = functions[0]->findPoint(axs);
    if (!points) {
        fprintf(stderr, "No load / store points found\n");
        return false;
    }

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

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

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

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

```



```

    return true;
}

void finishInstrumenting(BPatch_addressSpace *app, const char *newName) {
    BPatch_process *appProc = dynamic_cast<BPatch_process *>(app);
    BPatch_binaryEdit *appBin = dynamic_cast<BPatch_binaryEdit *>(app);
    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

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

    // Create/attach/open a binary
    BPatch_addressSpace *app = startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }
    // Instrument memory accesses
    if (!instrumentMemoryAccesses(app)) {
        fprintf(stderr, "instrumentMemoryAccesses failed\n");
        exit(1);
    }
    // Finish instrumentation
    const char *progName2 = "InterestingProgram - rewritten";
    finishInstrumenting(app, progName2);
}

```

8.3 Instrumenting a Function

```

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

```

```

#include <cstdlib>
#include <iostream>
#include <vector>

// Create an instance of class BPatch
BPatch bpatch;

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

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

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

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

    BPatch_image *appImage = app->getImage();

    // Scan for functions named "name"
    std::vector<BPatch_function *> functions;
    appImage->findFunction(name, functions);
}

```

```

    if (functions.size() == 0) {
        fprintf(stderr, "No function %s\n", name);
        return nullptr;
    }

    if (functions.size() > 1) {
        fprintf(stderr, "More than one %'s;' using the first one\n", name);
    }

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

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

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

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

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

    // Create the printf function call snippet
    std::vector<BPatch_snippet *> printfArgs;
    BPatch_snippet *fmt =
        new BPatch_constExpr("InterestingProcedure called %d times\n");
    printfArgs.push_back(fmt);
    BPatch_variableExpr *var = appImage->findVariable("myCounter");
    if (!var) {
        fprintf(stderr, "Could not find 'myCounter' variable\n");
        return false;
    }
    printfArgs.push_back(var);

    // Find the printf function

```

```

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

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

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

void finishInstrumenting(BPatch_addressSpace *app, const char *newName) {
    BPatch_process *appProc = dynamic_cast<BPatch_process *>(app);
    BPatch_binaryEdit *appBin = dynamic_cast<BPatch_binaryEdit *>(app);
    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

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

    // Create/attach/open a binary
    BPatch_addressSpace *app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        return EXIT_FAILURE;
    }
    // Find the entry point for function InterestingProcedure
    const char *interestingFuncName = "InterestingProcedure";

```

```

std::vector<BPatch_point *> *entryPoint =
    findPoint(app, interestingFuncName, BPatch_entry);
if (!entryPoint || entryPoint->size() == 0) {
    fprintf(stderr, "No entry points for %s\n", interestingFuncName);
    return EXIT_FAILURE;
}
// Create and insert instrumentation snippet
if (!createAndInsertSnippet(app, entryPoint)) {
    fprintf(stderr, "createAndInsertSnippet failed\n");
    return EXIT_FAILURE;
}
// Find the exit point of main
std::vector<BPatch_point *> *exitPoint = findPoint(app, "main", BPatch_exit);
if (!exitPoint || exitPoint->size() == 0) {
    fprintf(stderr, "No exit points for main\n");
    return EXIT_FAILURE;
}
// Create and insert instrumentation snippet 2
if (!createAndInsertSnippet2(app, exitPoint)) {
    fprintf(stderr, "createAndInsertSnippet2 failed\n");
    return EXIT_FAILURE;
}
// Finish instrumentation
const char *progName2 = "InterestingProgram - rewritten ";
finishInstrumenting(app, progName2);
}

```

8.4 Binary Analysis

```

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

#include <stdio>

// Create an instance of class BPatch
BPatch bpatch;

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

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

```

```

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

int binaryAnalysis(BPatch_addressSpace *app) {
    BPatch_image *appImage = app->getImage();
    int insns_access_memory = 0;
    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);
    if (functions.size() == 0) {
        fprintf(stderr, "No function InterestingProcedure\n");
        return insns_access_memory;
    } else if (functions.size() > 1) {
        fprintf(stderr,
            "More than one InterestingProcedure; using the first one\n");
    }
    BPatch_flowGraph *fg = functions[0]->getCFG();
    std::set<BPatch_basicBlock*> blocks;
    fg->getAllBasicBlocks(blocks);
    for (auto block_iter = blocks.begin(); block_iter != blocks.end();
        ++block_iter) {
        BPatch_basicBlock *block = *block_iter;
        std::vector<InstructionAPI::Instruction::Ptr> insns;
        block->getInstructions(insns);
        for (auto insn_iter = insns.begin(); insn_iter != insns.end();
            ++insn_iter) {
            InstructionAPI::Instruction::Ptr insn = *insn_iter;
            if (insn->readsMemory() || insn->writesMemory()) {
                insns_access_memory++;
            }
        }
    }
}

```

```

    return insns_access_memory;
}
int main() {
    // Set up information about the program to be instrumented
    const char *progName = "InterestingProgram";
    int progPID = 42;
    const char *progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;
    // Create/attach/open a binary
    BPatch_addressSpace *app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }
    int memAccesses = binaryAnalysis(app);
    fprintf(stderr, "Found %d memory accesses\n", memAccesses);
}

```

9 Optimization

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 it, 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.

9.1 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.

Since Dyninst 10.0.0, Dyninst supports parsing binaries in parallel, which significantly improve the analysis speed. We typically have about 4X speedup when analyzing binaries with 8 threads. By default, Dyninst will use all the available cores on your system. Please set environment variable `OMP_NUM_THREADS` to the number of desired threads.

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. Using these functions can result in a significant performance win when inserting instrumentation in many locations.

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.

9.2 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:

Save General Purpose Registers

In order to ensure that instrumentation doesn't corrupt the program, Dyninst saves all live general purpose registers.

Save Floating Point Registers

Dyninst may decide to separately save any floating point registers that may be corrupted by instrumentation.

Generate A Stack Frame

Dyninst builds a stack frame for instrumentation to run under. This provides the illusion to instrumentation that it is running as its own function.

Calculate Thread Index

Calculate an index value that identifies the current thread. This is primarily used as input to the Trampoline Guard.

Test and Set Trampoline Guard

Test to see if we are already recursively executing under instrumentation, and skip the user instrumentation if we are.

Execute User Instrumentation

Execute any `BPatch_snippet` code.

Unset Trampoline Guard

Marks the this thread as no longer being in instrumentation

Clean Stack Frame

Clean the stack frame that was generated for instrumentation.

Restore Floating Point Registers

Restore the floating point registers to their original state.

Restore General Purpose Registers

Restore the general purpose registers to their original state.

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

Avoid 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.

Disable recursive trampolines

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 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.

Don't save floating point registers

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 snippets 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.

Disable stack frames

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

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.

10 Common Pitfalls

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

Attach followed by detach

If a mutator attaches to a mutatee, and immediately exits, the current behavior is that the mutatee is left suspended. To make sure the application continues, call `detach` with the 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`.

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.

11 References

- B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications* (to appear), 2000.
- J. K. Hollingsworth and B. P. Miller, "Using Cost to Control Instrumentation Overhead," *Theoretical Computer Science*, 196(1-2), 1998, pp. 241-258.
- 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.
- 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.
- J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *PLDI*. June 18-21, 1995, La Jolla, CA, ACM, pp. 291-300.