Paradyn Parallel Performance Tools

DyninstAPI Programmer's Guide

Release 4.1
April 2004

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

1. INTRODUCTION

The normal cycle of developing a program is to edit source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing, and not have to re-compile, re-link, or even re-execute the program to change the binary. At first thought, this may seem like a bizarre goal, however there are several practical reasons we may wish to have such a system. For example, if we are measuring the performance of a program and discover a performance 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 running program. The API also permits changing or removing subroutine calls from the application program. Runtime 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 code patching. The API and a simple test application are described in [1]. This API is based on the idea of Dynamic Instrumentation described in [3].

The unique feature of this interface is that it makes it possible to insert and change instrumentation in a running program. This differs from other post-linker instrumentation tools [5] that permit code to be inserted into a binary before it starts to execute.

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

2. ABSTRACTIONS

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

[1] To generate more complex code, extra (initially un-called subroutines) can be linked into the application program, and calls to these subroutines can be inserted at runtime via this interface.

The API is designed so that a single instrumentation process can insert snippets into multiple processes executing on a single machine. To support multiple processes, two additional abstractions, threads and images, are included in the API. A *thread* refers to thread of execution. Depending on the programming model, a thread can correspond to either a normal process or a lightweight thread. *Images* refer the static representation of a program on disk. Images contain points where their code can be modified. Each thread is associated with exactly one image.

The 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 (currently only gcc on SPARC/Solaris), type checking is performed on code to be inserted into the mutatee. See Section 4.18.2 for a complete description of the type system.

3. SIMPLE EXAMPLE

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

A mutator program must create a single instance of the class BPatch. This object is used to access functions and information that are global to the library. It must not be destroyed until the mutator has completely finished using the library. For this example, we will assume that the mutator program has declared a global variable called bpatch of class BPatch.

The first thing a mutator needs to do is identify the application process to be modified. If the process is already in execution, this can be done by specifying the executable file name and process id of the application as arguments to create an instance of a thread object:

```
appThread = bpatch.attachProcess(name, proccesId);
```

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

```
appThread = bpatch.createProcess(pathname, argv);
```

Once the application thread has been created, the mutator defines the snippet of code to be inserted and the points where they should be inserted. For example, if we wanted to count the number of times a procedure called InterestingProcedure executes, the mutator might look like this:

```
BPatch_image *appImage;
BPatch_Vector<BPatch_point*> *points;
BPatch_Vector<BPatch_function *> functions;

// Open the program image associated with the thread and return a
//   handle to it.
appImage = appThread->getImage();

// fine and return the BPatch_function
appImage->findFunction(“InterestingProcedure”, functions);

// find and return the entry point to the "InterestingProcedure".
points = functions[0]->findPoint(BPatch_entry);

// Create a counter variable (but first get a handle to the correct type).
// by allocating in the application's address space.
BPatch_variableExpr *intCounter =
    appThread->malloc(*appImage->findType("int"));

// Create a code block to increment the integer by one.
//       intCounter = intCounter + 1
//
BPatch_arithExpr addOne(BPatch_assign, *intCounter,
        BPatch_arithExpr(BPatch_plus, *intCounter, BPatch_constExpr(1)));

// insert the snippet of code into the application.
appThread->insertBlock(addOne, *points);

// continue execution
appThread->continueExecution();

// wait for mutatee to finish while allowing Dyninst to handle events
while( !appThread->isTerminated() ){
    bpatch.waitForStatusChange();
}
```

4. INTERFACE

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

4.1 Class BPatch

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

```
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 first element of the array is `low`, and the last is `high`. The standard rules of type compatibility, described in Section 4.18.2 are used with arrays created using this function.

```
BPatch_type *createEnum(const char *name, BPatch_Vector<char *>
     elementNames, BPatch_Vector<int> elementIds)
BPatch_type *createEnum(const char *name, BPatch_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. The standard rules of type compatibility, described in Section 4.18.2 are used with enums created using this function.

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

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

```
BPatch_type *createStruct(const char *name, BPatch_Vector<char *>
     fieldNames, BPatch_Vector<BPatch_type *> fieldTypes)
```

Create a new structure type. The name of the structure is specified in the `name` parameter. The `fieldNames` and `fieldTypes` vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return NULL). The standard rules of type compatibility, described in Section 4.18.2 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 of the function creates a pointer whose size is the same size equal to `sizeof(void*)` on the target platform where the mutatee is running. In the second form of the command, the size of the pointer is the value passed in the `size` parameter.

```
BPatch_type *createUnion(const char *name, BPatch_Vector<char *>
     fieldNames, BPatch_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 standard rules of type compatibility, described in Section 4.18.2 are used with unions created using this function. The size of the union is the size of the largest element in the `fieldTypes` vector.

```
const char *getEnglishErrorString(int number)
```

This function returns the descriptive error string for the passed API error number. The returned string may contain placeholders (%s) to indicate that a parameter from the error callback (see the next section) should be substituted at that location.

```
BPatch_Vector<BPatch_thread*> *getThreads()
```

Return the list of threads that are currently defined. This list includes threads that were directly created by calling new on BPatch_thread, and indirectly by the UNIX fork or NT CreateProcess system call. *The creation of BPatch_thread objects for indirectly created threads is not yet implemented.*

`BPatch_thread *attachProcess(const char *path, int pid)` *not implemented on AIX*

`BPatch_thread *attachThread(const char *path, int pid, int tid)` *not yet implemented*

```
BPatch_thread *createProcess(const char *path, char *argv[],
     char *envp[] = NULL, int stdin_fd=0, int stdout_fd=1, int
     stderr_fd=2)
```

Each of these functions returns a pointer to a new instance of the BPatch_thread class. The “path” parameter needed by most of these functions should be the pathname of the executable file containing the thread’s code. The attachProcess function returns a `BPatch_thread` associated with an existing process. On some platforms, the path parameter can be NULL since the executable image can be derived from the process pid. The createThread function returns a new `BPatch_thread` associated with an existing thread within a process. The meaning of thread and process is implementation specific. The ability to use these two functions to create a `BPatch_thread` object for an existing process depends on support from the underlying operating system and may not be implemented on all platforms. A thread attached to using one of these functions is put into the stopped state. The createProcess function creates a new process and returns a new `BPatch_thread` 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.

`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 have 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.

`void setTrampRecursive (bool state)` *not implemented on Compaq Tru64 UNIX*

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, code will be created with recursion protection or not, depending on the current state of flag. Changing the flag is **not** retroactive, and inserting more snippets will not change recursion protection at the point. The recursion protection increases the overhead of instrumentation points, so if there is no way for the snippets to call themselves, then calling this method with the parameter `true` will result in a performance gain. This default value of this flag is `false`.

`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 turn type-checking back on is similar to type cast operation is the C programming language.

```
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.

4.1.1 Callbacks

The following functions are intended as a way for API users to be informed when a significant event occurs. Each allows a user to register a handler for some such event. The return code for all callback registration functions is the handler that was previously registered (which may be NULL if no handler has previously been registered).

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

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

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

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

This function registers the error callback function with the BPatch class. The return value is the previous error callback function. The error callback is explicitly registered (rather than using a pure a virtual function) so that BPatch users can change the error callback during program execution (i.e., one error callback before a GUI is initialized, and a different one after).

```
typedef void (*BPatchThreadEventCallback)(BPatch_thread *thread);
```

This is the prototype for most callback functions associated with events that occur in a thread. The `thread` parameter is the thread that the event has occurred in.

```
BPatchThreadEventCallback registerExecCallback(
     BPatchThreadEventCallback func)
```
implemented on Solaris, Linux, AIX, Compaq Tru64 UNIX, and Irix

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

```
BPatchThreadEventCallback registerThreadCreateCallback(
     BPatchThreadEventCallback func)
```
not yet implemented

Registers a function to be called when a new thread is created.

```
BPatchThreadEventCallback registerThreadDeleteCallback(
     BPatchThreadEventCallback func)
```
not yet implemented

Registers a function to be called when a new thread is terminated.

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

This is the prototype for the post fork callback, which is called after a fork. The `parent` parameter is the parent thread, and the `child` parameter is a `BPatch_thread` representing the newly created process. When invoked as a pre-fork callback, the child is NULL.

```
BPatchForkCallback registerPreForkCallback(
     BPatchForkCallback func)
```
not implemented on Windows (forking doesn't exist on Windows)

Registers a function to be called when a BPatch_thread forks a new process. This callback is invoked just before the fork is performed. When the callback is invoked, the thread performing the fork will be stopped.

```
BPatchPostForkCallback registerPostForkCallback(
     BPatchPostForkCallback func)
```
not implemented on Windows (forking doesn't exist on Windows)

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

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

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

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

```
BPatchThreadEventCallback registerExitCallback(
     BPatchExitCallback func)
```

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

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

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

```
BPatchThreadEventCallback registerDynLinkCallback(
     BPatchThreadEventCallback func)
```

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

4.2 Class BPatch_thread

The **BPatch_thread** class operates on (and creates) code in execution.

`BPatch_thread(BPatch_Vector<BPatch_thread&>threads)` *not yet implemented*

Creates a new "virtual" thread from a list of threads. This permits operations to be performed on several threads as a group. This can (potentially) increase the efficiently of the requests because they can be processed in parallel.

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

Return the executable file associated with this `BPatch_thread` object and return a handle to it. Depending on the implementation this might also parse the application's symbol table.

```
bool getLineAndFile (unsigned long addr, unsigned short& lineNo,
                        char* fileName,int length)
```
not implemented on Irix

Given the address, `addr`, lookup the line number and source file that contains the source code that corresponds to this location. This function returns true on a successful lookup, and false on a failure. Failures can be due to either invalid addresses being passed, or if the program was not compiled with debugging symbols. The first `length` characters of the source file name are copied into `fileName`.

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

These three functions change the running state of the thread. `stopExecution` puts the thread into a stopped state. Depending on the operating system, stopping one thread may stop all threads associated with a process. `continueExecution` continues execution of the thread (or group of threads if they have to be stopped atomically). `terminate-Execution` terminates execution of the thread and will result in the exit callback being called if registered. Each function returns true on success, or false for failure. Stopping or continuing a termiated thread will fail.

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

There three functions query the status of a thread. `isStopped` returns true if the thread 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. `isTer-minated` returns true if the thread has exited. Any of these functions may be called multiple times and calling them will not affect the state of the thread.

`void catchSignal(int signum)` *not yet implemented*
`void ignoreSignal(int signum)` *not yet implemented*

These two functions indicate that the process should be stopped or not when it receives the named signal.

`int dumpCore(const char *file, const bool terminate)` *implemented only on AIX*

This function causes the thread to dump its state to the passed file argument. If the `ter-minate` flag is true, the thread is also terminated. The ability to use this function depends on support from the underlying operating system and may not be implemented on all platforms.

`int dumpImage(const char *file)` *not implemented on NT*

This function causes the thread to write the in-memory version of the program to the specified file. **This function is not intended for general use, but rather to help debug implementations of dyninst. It's semantics and level of implementation varies greatly between platforms.**

`bool dumpPatchedImage(const char* file)`

This function causes the thread to write the in-memory version of the program to the specified file. This function **is intended for general use**. This produces a valid executable with the mutations in place. Mutated shared libraries are correctly saved on Sparc and Linux, but not AIX. *implemented only on AIX, Solaris and Linux*

```
void enableDumpPatchedImage()
```

This function must be called once before inserting instrumentation into the mutate that should be saved when `dumpPatchedImage` is called. Normally, this is called immediately after `createProcess.`

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

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

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

Free the memory in the passed ptr. The programmer is responsible to verify 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).

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

Retrieves a variable which exists in a child process and was inherited from and originally created in the parent process. This function is invoked on the child process's BPatch_thread which is created automatically when a fork occurs. The BPatch_thread for the child process can be retrieved via a BPatchForkCallback. Argument `parentVar` is a BPatch_variableExpr that was created in a parent process with BPatch_thread::malloc(). If it is determined that `parentVar` wasn't allocated in the parent process, then NULL is returned.

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

Allows one to retrieve a handle to snippet which exists in a child process and was inherited from and originally created in the parent process. This function is invoked on the child process's BPatch_thread which is created automatically when a fork occurs. The BPatch_thread for the child process can be retrieved via a BPatchForkCallback. Argument `parentSnippet` is a BPatchSnippetHandle for a BPatch_snippet that was inserted into the parent process. If it is determined that `parentSnippet` isn't associated with the parent process, then NULL is returned.

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

Cause snippet to be evaluated once at the next available opportunity. This interface is useful to cause an initialization function to be called in the application. The process must be stopped to call this function. The behavior is synchronous, that is the oneTimeCode function won't return until after the snippet has been run in the application.

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

Insert a snippet of code at the specified point. If a list of points is supplied, insert the code snippet at each point in the list. The `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. 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` can be used to indicate that the snippet should be inserted before or after all such snippets, respectively.

The semantics of `BPatch_callBefore` and `BPatch_callAfter` when applied to entry and exit points are still being fully implemented. The following table summarizes the intention of each point:

BPatch_procedureLocation	**BPatch_callWhen**	**Meaning**
BPatch_entry	BPatch_callBefore	First instruction in subroutine
BPatch_entry	BPatch_callAfter	First instruction in subroutine after activation record (local variables) have been created
BPatch_exit	BPatch_callBefore	Last instruction in subroutine before activation record (local variables) destroyed
BPatch_exit	BPatch_callAfter	Last instruction in subroutine

Currently the two combinations to allow access just before and after the local variables have been created are not implemented.

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

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

`bool removeFunctionCall(BPatch_point &point)`

Disable the function call at the specified location. The point specified must be a valid call point in the image of the requesting thread. 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 left to the library implementor, but might include branching over the call, or replacing it with nops. (Parameters are still evaluated).

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

Replace all calls to 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`. Return true upon success, false otherwise. *implemented on SPARC Solaris, X86 Linux, X86 Windows, and Compaq Tru64 UNIX.*

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

The function call at the specified point is changed to be a call 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.

Note: 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.

`void setInheritSnippets(bool inherit)` *not yet implemented*

Set a flag to indicate if instrumentation snippets should be inherited when the thread forks. By default, instrumentation snippets are inherited by the child process.

`void setMutationsActive(bool)`

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

`void detach(bool cont)`

Detach from the thread. The thread must be stopped to call this function. The cont parameter is used to indicate if the thread should be continued as a result of detaching.

```
int getPid()
```

Return the id of the process to which the thread belongs.

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

```
BPatch_exitType terminationStatus()
```

If the process has exited, will indicate whether the process exited normally or because of a signal. If the process hasn't exited, NoExit will be returned.

```
int getExitCode()
```

If the process exited in a normal way, will return the associated exit code.

```
int getExitSignal()
```

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

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

Load a dynamically linked library into the thread's address space. The libname parameter identifies 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 true if the library was loaded successfully, otherwise it returns false. If reload is true, then the library will be reloaded *at startup* by the rewritten binary produced by a call to BPatch_thread::dumpPatchedImage.

```
~BPatch_thread()
```

In addition to cleaning up its own state, the BPatch_thread class destructor may also kill the underlying thread or process. If the process was **created** by using a BPatch_thread constructor (as opposed to attaching to an existing thread by passing a pid to the constructor), and detach was not called before the destructor then the process will be terminated by the destructor. Otherwise it will continue to execute **and any inserted snippets will remain installed.**

One additional convenience (non-member) function is provided to test if the status of any of the threads managed by the mutator has changed.

4.3 Class BPatch_sourceObj

The BPatch_sourceObj class is the parent class for the BPatch_function, BPatch_module, and BPatch_image classes. It provides a set of common methods for all three classes. In addition, it can be used to build a "generic" source navigator using the getObjParent and getSourceObj methods to get parents and children of a given level (i.e. the parent of a module is an image, and the children will be the functions).

`BPatch_sourceType getSrcType()`

Return the type of the current source object. Currently, the following values are available BPatch_sourceProgram, BPatch_sourceModule, BPatch_sourceFunction, and BPatch_-sourceUnknown_type. Eventually, the following additional types will be available: BPatch_sourceOuterLoop, BPatch_sourceLoop, BPatch_srcBlock, BPatch_sourceState-ment

`void getSourceObj(BPatch_Vector<BPatch_sourceObj *> &objs)`

Return the children source objects of the current source object.

`BPatch_sourceObj *getObjParent()`

Return the parent source object of the current source object. The parent of a BPatch_-image is NULL.

```
BPatch_Vector<BPatch_variableExpr *> *findVariable(const char
     *name)
```

Lookup and return 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_language getLanguage()`

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

`BPatch_type *getType(char *name)`*not yet implemented*

Lookup and return a handle to the named type. The handle can be used as an argument to malloc to create new variables of the corresponding type.

`void getCallStack(BPatch_Vector<BPatch_frame>& stack)`

This function fills the given vector with current information about the call stack of the thread. Each stack frame is represented by a BPatch_frame (see section 4.17 for information about this class).

4.4 Class BPatch_function

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

```
bool getLineAndFile(int &start, int &end, char *filename,
     int &max)
```
not implemented on Irix

> This function returns the (approximate) line number range for the specified function. It returns false the function does not have line number information (i.e., stripped or compiled without debugging). The line number of the first executable statement of a function is return in `start.` The line number of the last executable statement of a function is return in `end.` Up to max characters of the source file name for the function is copied into `filename`, and max is updated to indicate the number of characters actually copied.

```
bool getLineToAddr(unsigned short lineNo,
     BPatch_Vector<unsigned long>& buffer,
     bool exactMatch = true)
```
not implemented on Irix

> Return the address(es) of the instructions that represent the code for the passed line number, `lineNo`. If `exactMatch` is true, require and exact match for the passed line number. The function returns true on a successful lookup, and false if the line number can not be located or if the program does no contain debugging information.

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

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

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

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

```
BPatch_Vector<BPatch_localVar *> *getParams()
```

> Return a vector of `BPatch_localVar` that contain the parameters for 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 be used in snippet expressions. NOTE: Using parameter `BPatch_localVar` expressions in snippets is only supported for parameters that have a position on the function's activation record. Parameters passed in registers (that remain in registers) cannot be accessed using this method yet.

```
BPatch_type *getReturnType()
```

> Return the type of the return value for this function.

`bool isInstrumentable()`

Returns true if the function can be instrumented, and false if it cannot. Various conditions can cause a function to be uninstrumentable. For example, on some platforms functions smaller than some specific number of bytes cannot be instrumented.

`bool isSharedLib()`

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

`bool isLib()`*not yet implemented*

This function returns true if the function is defined in a library (regardless of whether the library is shared or non-shared).

`const char *libraryName()`*not yet implemented*

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

`Bpatch_module *getModule()`

Return the module that defines this function. Depending on whether the program was compiled for debugging or the symbol table stripped, this information may not be available.

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

Copy the name of the module that contains this function into the buffer pointed to by `name`. Copy at most `maxLen` characters.

```
const BPatch_Vector<BPatch_point *> *findPoint(const
     BPatch_procedureLocation loc)
```

Return the BPatch_point or list of BPatch_points associated with the procedure. The BPatch_procedureLocation argument is one of BPatch_entry, BPatch_exit, BPatch_subroutine, BPatch_longJump, or BPatch_allLocations. 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. BPatch_longJumps returns any long jump statements made by the procedures. If the lookup fails to locate any points of the requested type, a list with zero elements is returned. *The BPatch_longJump location is not yet implemented.*

```
BPatch_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. Used primarily for memory access instrumentation. The BPatch_opCode is an enumeration of instruction types that may be requested: BPatch_opLoad, BPatch_opStore, and BPatch_opPrefetch. Any combination of these may be requested by passing an appropriate argument set containing the desired types.

The instrumentation points created by this function have additional memory access information attached to them. This allows such points to be used for memory access specific snippets (e.g. effective address). The memory access information attached is described under Memory Access classes elsewhere in this document.

`void *getBaseAddr()`

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

`unsigned int getSize()` *not yet implemented on Alpha*

Returns the size of the function in bytes.

`BPatch_flowGraph *getCFG()`

Returns the control flow graph for the function, or NULL if this information is not available.

4.5 Class BPatch_point

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

`BPatch_procedureLocation getPointType()`

Return the type of the point. This returned type is one of BPatch_entry, BPatch_exit, BPatch_subroutine, BPatch_longJump, or BPatch_address.

`BPatch_function *getCalledFunction()`

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

`void *getAddress()`

Returns the address of the first instruction at this point.

`int getDisplacedInstructions(int maxSize, void **insns)`

Copy (up to maxSize bytes), the instructions to be relocated at this point into the passed array (insns). Return the actual number of bytes of instructions copied.

`bool usesTrap_NP()`

Returns true if inserting instrumentation at this point requires using a trap. On the x86 architecture, because instructions are of variable size, the instruction at a point may be too small for the API library to replace it with the normal code sequence used to call instrumentation. Also, when instrumentation is placed at points other than subroutine entry, exit, or call points, traps may be used to ensure the instrumentation fits. In this case, the API replaces the instruction with a single-byte instruction that generates a trap. A trap handler then calls the appropriate instrumentation code. Since this technique is used only on some platforms, on other platforms this function always returns false.

```
const MemoryAccess* getMemoryAccess()
```

Returns the memory access object associated with this point.

```
const BPatch_Vector<BPatchSnippetHandle *> getCurrentSnippets()
const BPatch_Vector<BPatchSnippetHandle *>
                         getCurrentSnippets(BPatch_callWhen when)
```

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

4.6 Class BPatch_image

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

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

Return an instrumentation point at the specified address. This function is designed to permit users who wish to insert instrumentation at an arbitrary place in the code segment. Currently the implementation of this function may use a trap instruction, making these points more expensive than most instrumentation points. Also, on x86 platforms, users should take care to ensure that the requested point is not in the middle of a multi-byte instruction. The second form also returns the alternative BPatch_point in the given buffer if the new allocation overlaps another already existing BPatch_point object. That is, the existing BPatch_point object is assigned to alternative buffer and NULL is returned.

```
const BPatch_Vector<BPatch_function *> *getProcedures()
```

Return a table of the procedures in the image.

```
Const BPatch_Vector<BPatch_module *> *getModules()
```

Return a table of the modules in the image.

```
BPatch_Vector<BPatch_function*> *findFunction(const char *name,
     BPatch_Vector<BPatch_function*> &funcs)
```

Return a vector of `BPatch_function`’s for the function `name` defined, or NULL if the function does not exist. If *name* contains a POSIX-extended regular expression, a regex search will be performed on function names, and matching Bpatch_functions returned. Note: the BPatch_Vector argument *funcs* must be declared fully by the user before calling this function – passing in an uninitialized reference will result in undefined behavior.

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*> *
findFunction( BPatch_Vector<BPatch_function*> &funcs),
     BpatchFunctionNameSieve bpsieve, void *sieve_data)
```

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

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

typedef bool (**BpatchFunctionNameSieve*) (const char **name*, void **sieve_data*);

```
const BPatch_point *findLinePoint(const char *fileName, int line)
```
not yet implemented

Return the handle to the instrumentation point nearest to the requested fileName and line number. The nearest point to a requested line is the last executable instruction before the line (Note this function can have strange interactions with optimized code).

```
const BPatch_variableExpr *findVariable(const char *name)
const BPatch_variableExpr *findVariable(const BPatch_point
     &scope,
     const char *name)
```
second form of this method is not implemented on NT or MIPS/Irix.

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

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

Lookup and return a handle to the named type. The handle can be used as an argument to malloc to create new variables of the corresponding type.

`const char *getUniqueString()` *not yet implemented*

Lookup and return 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 is implementation specific and defined to have no semantic meaning.

```
bool getLineToAddr (const char* fileName,unsigned short lineNo,
     BPatch_Vector<unsigned long>& buffer,
     bool exactMatch = true)
```
not implemented on Irix

Return the address(es) of the instructions that represent the code for the passed line number, `lineNo`, in the passed source code file name, `fileName`. If `exactMatch` is true, require and exact match for the passed line number. The function returns true on a successful lookup, and false if the line number or source file cannot be located or if the program does not contain debugging information or the source file does not contain the line passed.

4.7 Class BPatch_module

An object of this class represents a program module, which is part of a program's executable image. BPatch_module objects are obtained by calling the BPatch_image member function getModules().

```
BPatch_Vector<BPatch_function*> *findFunction(const char *name,
     BPatch_Vector<BPatch_function*> &funcs)
```

Return a vector of `BPatch_function`'s for the function `name` defined, or NULL if the function does not exist. If *name* contains a POSIX-extended regular expression, a regex search will be performed on function names, and matching BPatch_functions returned. Note: the BPatch_Vector argument *funcs* must be declared fully by the user before calling this function – passing in an uninitialized reference will result in undefined behavior.

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

```
BPatch_function *findFunctionByMangled (const char *mangled_name)
```

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

```
bool getLineToAddr (unsigned short lineNo,
     BPatch_Vector<unsigned long>& buffer,
     bool exactMatch = true)
```
not implemented on Irix

Return the address(es) of the instructions that represent the code for the passed line number, `lineNo`. If `exactMatch` is true, require and exact match for the passed line number. The function returns true on a successful lookup, and false if the line number can not be located or if the program does no contain debugging information.

`const BPatch_Vector<BPatch_function *> *getProcedures()`

Return a table of the procedures in the module.

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

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

`const char *libraryName()` *not yet implemented*

Return the name of the library that contains the module. If the module is not part of a library, a NULL will be returned.

`Bool isSharedLib()`

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

`Bool isLib()` *not yet implemented*

This function returns true if the module is part of a library (regardless of whether the library is shared or non-shared).

`const char *getUniqueString()` *not yet implemented*

Lookup and return 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 is implementation specific and defined to have no semantic meaning.

4.8 Class BPatch_snippet

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

`const BPatch_type *getType()`

Return the type of the snippet.

`float getCost()`

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

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

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

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

Operator	**Description**
BPatch_assign	assign the value of `rOperand` to `lOperand`
BPatch_plus	add `lOperand` and `rOperand`
BPatch_minus	subtract `rOperand` from `lOperand`
BPatch_divide	divide `rOperand` by `lOperand`
BPatch_times	multiply `rOperand` by `lOperand`
BPatch_mod	compute the remainder of dividing `rOperand` into `lOperand` *Not yet implemented.*
BPatch_ref	Array reference of the form `lOperand[rOperand]`
BPatch_seq	Define a sequence of two expressions (similar to comma in C)
BPatch_min	Return the smaller of two operands *Not yet implemented.*
BPatch_max	Return the larger of two operands *Not yet implemented.*

`BPatch_arithExpr(BPatch_unOp, const BPatch_snippet &operand)`

Define a snippet consisting of a unary operator. The available unary operators are BPatch_negate, BPatch_addr, and Bpatch_deref. BPatch_negate takes an integer snippet and returns the negation of the snippet. BPatch_addr takes a variable reference snippet and returns a pointer to it. This is equivalent to the C operator (&) and is useful for call-by-reference parameters. Bpatch_deref takes a variable that is a pointer and de-references it. It is the equivalent of the C operator (*) and is useful for directly computing addresses of stored data.

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

Define a relational snippet. The available operators are:

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

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

`BPatch_breakPointExpr()`

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

`BPatch_bytesAccessedExpr ()`

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

`BPatch_constExpr(int value)`
`BPatch_constExpr(float value)` *not yet implemented*
`BPatch_constExpr(const char *value)`
`BPatch_constExpr(const void *value)`
`BPatch_constExpr(bool value)` *not yet implemented*

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

`BPatch_effectiveAddressesExpr ()`

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

`BPatch_funcJumpExpr (const BPatch_function &func)` *not implemented on IRIX*

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

When `func` returns, control flows as a return from the function in which this snippet is inserted.

```
BPatch_funcCallExpr(const BPatch_function& func,
     const BPatch_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. 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).

`BPatch_gotoExpr(const BPatch_gotoExpr &target)` *not yet implemented*

Branch to the passed snippet. When used with `BPatch_ifExpr`, the goto expression can be used for simple looping. To implement the C loop:

```
repeat
   i++
until (i == 50);
```

the following BPatch code would be used:

```
// addOne: i++   -- Add one to the intCounter (i), also create “label”
//    add One
BPatch_arithExpr addOne(BPatch_assign, *intI,
      BPatch_arithExpr(BPatch_plus, *intI, BPatch_constExpr(1)));

// if (i != 50) goto addOne
//   First definition is the boolean expression.
//   The second, generates the goto and the if statement
BPatch_boolExpr testFlag(BPatch_ne, *intI, BPatch_constExpr(50));
BPatch_ifExpr loopDone(testFlag, BPatch_gotoExpr(addOne));
```

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

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

`BPatch_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 be changed during subroutine execution, this snippet type is only valid at points that are entries to subroutines, or when inserted at a call point with the `when` parameter set to `BPatch_callBefore.`

`BPatch_pidExpr()` *not yet implemented*

This snippet results in an integer expression that contains the id of the process in which it is executing.

`BPatch_retExpr()`

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

`BPatch_sequence(const BPatch_Vector<BPatch_snippet*> &items)`

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

`BPatch_tidExpr()` *not yet implemented*

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

`BPatch_nullExpr()`

Defines a null snippet. This snippet contains no executable statements; however it is a useful place holder for the destination of a goto. For example, using goto and a nullExpr a while loop can be constructed. For example, to construct the while loop:

```
while (i  < 3) {
    i++;
}
```

The following snippets should be created:

```
BPatch_nullExpr loopDone;

// if (i > 3) goto loopDone
//   First definition is the boolean expression.
//   The second, generates the goto and the if statement
BPatch_boolExpr testFlag(BPatch_gt, *intI, BPatch_constExpr(3));
BPatch_ifExpr test(testFlag, BPatch_gotoExpr(loopDone));

// i++
BPatch_arithExpr addOne(BPatch_assign, *intI,
      BPatch_arithExpr(BPatch_plus, *intI, BPatch_constExpr(1)));

BPatch_Vector<BPatch_snippet *> statements;

statements.push_back(&test);
statements.push_back(&addOne);
statements.push_back(&loopDone);
```

```
        BPatch_sequence whileLoop(statements);
```

4.9 Class BPatch_type

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

`BPatch_Vector<BPatch_field *> *getComponents()`

> Returns a vector of the types of the fields in a BPatch_struct or BPatch_union. If the data class of the type is not BPatch_struct or BPatch_union, a null value is returned.

`BPatch_Vector<BPatch_cblock *> *getCblocks()`

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

`BPatch_type *getConstituentType()`

> Return the type of the base type. For a `BPatch_array` this is the type of each element, for a `BPatch_pointer` this is the type of the object the pointer points to. For `BPatch_typedef` types, this is the original type. For all other types, an undefined results will be returned.

`BPatch_dataClass getDataClass()`

> Returns the data class of the type.

```
const char *getLow()
const char *getHigh()
```

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

`const char *getName()`

> Return the name of the type.

`bool isCompatible(const BPatch_type &otype)`

> Returns true if the passed type is type compatible with this type. The rules for type compatibility are given in Section 4.18.2. If the two types are not type compatible, the error reporting callback function will be invoked one or more times with additional information about why the types are not compatible.

4.10 Class BPatch_variableExpr

The **BPatch_variableExpr** class is another class derived from snippet. It represents a variable or area of memory in a thread's address space. A `BPatch_variableExpr` can be obtained from a `BPatch_thread` using the `malloc` member function, or from a `BPatch_image` using the `findVariable` member function. BPatch_variableExpr provides two member functions not provided by other types of snippets:

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

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

```
bool writeValue(void *src, bool saveWorld=false)
void writeValue(void *src, int size, bool saveWorld=false)
```

Changes the value of the variable in an application's address space that is represented by this BPatch_variableExpr. The `src` parameter should point to a value of the variable's type. If the size parameter is supplied, then the number of bytes it specifies will be written. For the first version of this method, if the size of the variable is known (i.e., no type information) information, no data is copied and the method returns false. If saveWorld is true the value written by this function will be restored after restarting a mutatee produced by calling BPatch_thread::dumpPatchedImage(). Note that the value restored is the value in src, not the value in the mutatee when dumpPatchedImage() is called

```
void *getBaseAddr()
```

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

```
BPatch_Vector<BPatch_variableExpr *> getComponents()
```

Returns a vector of the components of a struct, or union. Each element of the vector is one field of the composite type, and contains a variable expression for accessing it.

4.11 Class BPatch_flowGraph

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

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

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

`void getEntryBasicBlock(BPatch_Vector<BPatch_basicBlock*>&)`

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

`void getExitBasicBlock(BPatch_Vector<BPatch_basicBlock*>&)`

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

`void getLoops(BPatch_Vector<BPatch_basicBlockLoop*>&)`

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

`void getOuterLoops(BPatch_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.

`void createSourceBlocks()`

Finds and stores the source code line information for each basic block in the control flow graph. That is for each basic block in the control flow graph, this method finds the source file(s) and line(s) that correspond to the instructions in that basic block.

`void fillDominatorInfo()`

Calculates and stores the immediate dominator tree information for the control flow graph. This method stores the dominator information in the basic block objects of the control flow graph.

(**Note:** If a function has a case statement or multi-jump instructions, the targets of the jumps are found by searching instruction patterns (peep-hole). The instruction patterns generated are compiler specific and we included in control flow graph generation the ones we have seen. During the control flow graph generation, if a pattern that is not handled is used for case statement or multi-jump instructions in the function address space, the generated control flow graph may not be complete.)

4.12 Class BPatch_loopTreeNode

The **BPatch_loopTreeNode** class provides a tree interface to a collection of instances of class `BPatch_basicBlockLoop contained` in a `BPatch_flowGraph.` The structure of the tree follows the nesting relationship of the loops in the flow graph. The root

`BPatch_loopTreeNode` instance has a null loop member since a function may contain multiple outer loops, the outer loops are contained in the root instance's vector of children. Each instance of `BPatch_loopTreeNode` is given a name which indicates its position in the hierarchy of loops.

`BPatch_basicBlockLoop *loop`

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

`BPatch_Vector<BPatch_loopTreeNode *> children`

The tree nodes for the loops nested under this loop.

`const char *name()`

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

4.13 Class BPatch_basicBlock

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

`void getSources(BPatch_Vector<BPatch_basicBlock*>&)`

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

`void getTargets(BPatch_Vector<BPatch_basicBlock*>&)`

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

`bool dominates(BPatch_basicBlock*)`

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

`BPatch_basicBlock* getImmediateDominator()`

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

`void getImmediateDominates(BPatch_Vector<BPatch_basicBlock*>&)`

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

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

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

`void getSourceBlocks(BPatch_Vector<BPatch_sourceBlock*>&)`
not implemented only on Irix

Fill the given vector with a list of source blocks contributing to this basic block’s instruction sequence.

`int getBlockNumber()`

Return the ID number of this basic block. The IDs 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.

`bool getAddressRange(void*& _startAddress, void*& _endAddress)`

This function returns the starting and ending addresses of the range of instructions that make up this basic block, if this information is available. It returns true if the returned addresses are valid, and false

4.14 Class BPatch_basicBlockLoop

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

`void getBackEdges(BPatch_Vector<BPatch_basicBlock*>&)`

Fill the given vector with a list of pointers to the basic blocks that are the sources of the back edges that define the natural loop.

`void getContainedLoops(BPatch_Vector<BPatch_basicBlockLoop*>&)`

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

`void getOuterLoops(BPatch_Vector<BPatch_basicBlockLoop*>&)`

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

`void getLoopBasicBlocks(BPatch_Vector<BPatch_basicBlock*>&)`

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

```
void getLoopBasicBlocksExclusive(
     BPatch_Vector<BPatch_basicBlock*>&)
```

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

`BPatch_basicBlock* getLoopHead()`

Return the basic block at the head of this loop.

`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).

`BPatch_Set<BPatch_variableExpr*>* getLoopIterators()` *not yet implemented*

Return a set containing the variables used as loop iterators.

4.15 Class BPatch_sourceBlock

An object of this class represents a source code level block. Each source block objects consists of a source file and 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. That is 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. *not implemented on Irix.*

`const char* getSourceFile()`

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

`void getSourceLines(BPatch_Vector<unsigned short>&)`

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

4.16 Class BPatch_cblock

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

`BPatch_Vector<BPatch_field *> *getComponents()`

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

`BPatch_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.

4.17 Class BPatch_frame

A **BPatch_frame** object represents a stack frame. The getCallStack() member function of BPatch_thread returns a vector of BPatch_frame objects representing the frames currently on the stack.

`BPatch_frameType getFrameType()`

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

Frame Type	Meaning
BPatch_frameNormal	A normal stack frame.
BPatch_frameSignal	A frame that represents a signal invocation. *This type is implemented only on Sparc, x86, IA-64, and AIX platforms.* On other platforms, calls to getCallStack() for a thread that is inside a signal handler return results that are undefined.
BPatch_frameTrampoline	A frame the represents a call into instrumentation code. *This type is only implemented on x86, IA-64, and AIX.* On other platforms, calls to getCallStack() for a thread that is inside instrumentation code return results that are undefined.

`void *getFP()`

Returns the frame pointer for the stack frame. This information is not available on Windows NT.

`void *getPC()`

Returns the program counter associated with the stack frame.

`BPatch_function *findFunction()`

Returns the function associated with the stack frame.

`int getSignalNumber()` *not yet implemented*

If the frame represents a signal delivery (getFrameType() returns BPatch_frameSignal), this function returns the number of the signal.

`BPatch_point *findPoint()` *not yet implemented*

Returns a BPatch_point() representing the location of the program counter for the frame.

4.18 Container Classes

4.18.1 Class BPatch_Vector

The **BPatch_Vector** class is a container used to hold other objects used by the API. It is based on the Standard Template Library (STL) Vector container class. At the time of the writing of this document, STL has been adopted as part of the ANSI C++ standardization, but implementations were not widely available. As a result, the initial version of the API uses its own compatible subset of the Vector class.

`BPatch_Vector()`

Create a new empty vector.

`int size()`

Return the number of elements in the container instance.

`void push_back(const T& x)`

Add x to the end of the Vector.

`void clear()`

Removes all elements from the Vector.

`const T& operator[](int n) const`

Return the nth element of the Vector.

The following example illustrates how to declare a vector, add elements to it, and iterate over them:

```
BPatch_Vector<int> list_of_ints;

list_of_ints.push_back(1);
list_of_ints.push_back(2);

for (int i = 0; i < list_of_ints.size(); i++)
    printf(“%d\n”, list_of_ints[i]);
```

4.18.2 Class BPatch_Set

BPatch_Set is another container class, similar to the set class in the Standard Template Library (STL). It maintains a collection of objects and provides fast lookup. Elements are ordered by a comparison function, which can be user-supplied. This allows for efficiently returning a sorted list of elements, or returning the value of the minimum or maximum element.

`BPatch_Set()`

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

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

Copy constructor.

`int size()`

Returns the number of elements in the set.

`bool empty()`

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

`void insert(const T&)`

Insert the given element into the set.

`void remove(const T&)`

Remove the given element from the set.

`bool contains(const T&)`

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

`T* elements(T*)`

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

`T minimum()`

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

`T maximum()`

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

`BPatch_Set<T,Compare>& operator= (const BPatch_Set<T,Compare>&)`

The assignment operator.

`bool operator== (const BPatch_Set<T,Compare>&)`

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

`bool operator!= (const BPatch_Set<T,Compare>&)`

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

`BPatch_Set<T,Compare>& operator+= (const T&)`

Add the given object to the set.

`BPatch_Set<T,Compare>& operator|= (const BPatch_Set<T,Compare>&)`

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

`BPatch_Set<T,Compare>& operator&= (const BPatch_Set<T,Compare>&)`

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

`BPatch_Set<T,Compare>& operator-= (const BPatch_Set<T,Compare>&)`

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

`BPatch_Set<T,Compare> operator| (const BPatch_Set<T,Compare>&)`

Set union operator.

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

Set intersection operator.

`BPatch_Set<T,Compare> operator- (const BPatch_Set<T,Compare>&)`

Set difference operator.

4.19 Memory Access Classes

Instrumentation points created through findPoint(const BPatch_Set<BPatch_opCode>& ops) get memory access information attached to them. This information is used by the memory access snippets, but is also available to the API user. It is however machine dependent, thus non-portable. The classes that encapsulate memory access information are contained in the BPatch_memoryAccess_NP.h header.

4.19.1 Class BPatch_memoryAccess

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

`bool isALoad_NP()`

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

`bool isAStore_NP()`

Returns true if memory is written. Note that some machine instructions may both load and store, for instance CAS (compare and swap) on SPARC.

`bool isAPrefetch_NP()`

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

`short prefetchType_NP()`

If the memory access is a prefetch, this method returns a platform specific prefetch type. *On SPARC this returns the prefetch type as encoded in the instruction. See the SPARC Architecture Manual (version 9) for details.*

`BPatch_addrSpec_NP getStartAddr_NP()`

Returns an address specification that allows effective address to be computed.

`BPatch_countSpec_NP getByteCount_NP()`

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

4.19.2 Class BPatch_addrSpec_NP

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

`int getImm()`

Returns 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. 0 < i < 2 in this release. Register numbers are positive; a value of -1 means no register.

4.19.3 Class BPatch_countSpec_NP

This class encapsulates the information required to determine the number of bytes transferred by a memory access. In this release it is alias for BPatch_addrSpec; do not rely on this implementation, it may change in future releases.

4.20 Type System

The dyninst API 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, the dyninst API will verify the type compatibility of the operations performed by the mutator.

The rules for type computability are that two type 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 type to be compatbible:

`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 they their return types are compatible, 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 each 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_unkownType, 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_unkownType.

5. USING THE API

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

5.1 Overview of Major Steps

To use the dyninstAPI, you just have to:

(1) *Create a mutator program (Section 5.1):* You need to create a program that will modify some other program. For example, the mutator shown in Section 6.

(2) *Set up your mutatee (Section 5.3):* On some platforms, you need to link your application with the dyninstAPI's run time instrumentation library. Note: this step is only needed in the initial release of API. Future releases will eliminate this restriction.

(3) *Run the mutator (Section 5.4):* the mutator will either create a new process or attach to an existing one (depending on the whether createProcess or attachProcess is used).

Sections 5.2 through 5.4 explain these steps in more detail. In addition, Section 5.5 describes any issues related to a specific hardware or operating systems. In this section, we assume that you have already installed the API distribution and setup the `PLATFORM` and `DYNINST_ROOT` environment variables. The installation of the API is described in the README file in the distribution tar file.

5.2 Creating a Mutator Program

The first step in using the dyninstAPI 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 mutatee is the sample program (/) given in Section 6. The following fragment of a Makefile shows how to link your mutator program with the dyninstAPI library on most platforms:

```
retee.o: retee.c
$(CC) -c $(CFLAGS) -I$(DYNINST_ROOT)/core/dyinstAPI/h \
          retee.c

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

On Solaris and Linux, the option "-lelf" must also be added to the link step. On Linux, the option "-ldwarf" must also be added to the link step. On Solaris, the option "-lstdc++" must be added to the link step. On Compaq Tru64 UNIX, the option "-lmld" must also be supplied. On AIX, the option –lbsd must also be added to the link step. On MIPS, the options "-ldwarf" and "–lelf" must be added to the link step. You will also need to make sure that the LD_LIBRARY_PATH or LIBPATH (AIX) environment variable includes the directory that contains the dyninst shared library. This is typically $DYNINST_ROOT/lib/$PLATFORM.

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

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

```
     CC = cl

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

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

5.3 Setting Up your Application Program (mutatee)

On most platforms, you can instrument unmodified binary (a.out) files. However, there is a base shared library that needs to be available to be loaded into your application (by the mutator), and you may wish to create library of pre-compiled instrumentation routines that you mutator will insert calls to.

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) must be linked with your application. Simply add these files to the line **<insert any additional modules here>** in Figure 1. On SPARC Solaris, AIX, Linux, and Compaq Tru64 UNIX, you may put such code into a dynamically loaded shared library, which your mutator program can load into the mutatee at runtime using the loadLibrary member function of BPatch_thread.

Additionally, on most platforms we need to use the flags `-g` (to generate debugging) when compiling. The command line switches used to specify these options are different for Visual C++ on Windows NT; see section 5.5.2 for information about compiling on Windows NT.

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:

$DYNINST_ROOT/lib/$PLATFORM/libdyninstAPI_RT.so.1

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

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

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

```
OBJECTS = main.o this.o that.o

LIBDIR = $DYNINST_ROOT/lib/$PLATFORM

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

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

Figure 1: Changing Your Makefile to Link an Application as a dyninstAPI mutatee. Note: some platforms require a few additional options; see Section 5.5.

5.4 Running Your 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 Section 6:

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

5.5 Architectural Issues

Certain platforms require slight modifications to the procedures discussed above. In this subsection, we describe each of them in turn.

5.5.1 Solaris

When using the Sun C or Fortran compilers, you should also specify the **`-xs`** option together with **`-g`**. The **`-g`** option alone will direct the compiler to place debugging information in the object files (**`.o`** files), but it will not place the debugging information on the executable (**`a.out`**) file. You must use the **`-xs`** option so that the compiler will add the debugging information to the a.out file. The **`-xs`** option is not needed if you are using gcc. The following is an example of linking on Solaris.

```
OBJECTS = main.o this.o that.o
LIBDIR = $DYNINST_ROOT/lib/$PLATFORM
bubba.pd: ${OBJECTS}
            cc -g -xs \
            ${OBJECTS} \
            -lm -lcurses -ltermcap \
            -o bubba
```

Linking an application to run with the dyninstAPI.
Items in **Bold face** show the changes for Solaris.

Figure 2: Sample Makefile for Solaris

5.5.2 Windows NT

Under Windows NT, the insertion of code at some instrumentation points requires the use of an interrupt instruction, which generates an event that must be serviced by the mutator process. The API library performs this event handling transparently in the calls `pollForStatusChange` and `waitForStatusChange`. This means that it is important under Windows NT to call one of these functions frequently, in order to ensure that the events are handled in a timely manner. It also means that a mutator program cannot detach or exit and leave instrumentation code running in the mutatee, since there would then be no program to handle the interrupt events.

On Windows NT the run-time instrumentation library is loaded dynamically, and you **do not** need to relink your application with this library. First the environment variable `DYNINSTAPI_RT_LIB` is checked; if it is defined, the library is loaded from this file. If the variable is not defined, Windows looks for a DLL named `libdyninstAPI_RT.dll` using its DLL search strategy. That strategy looks for DLLs in the following order:

1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory (usually C:\WINDOWS\SYSTEM32).
4. The directories that are listed in the PATH environment variable.

The API now requires a recent version of the DbgHelp DLL to run. This DLL ships with Windows 2000 and Windows XP, but the version shipped with Windows 2000 does not include some of the functionality required by the API. An up-to-date version of this DLL is included with the binary release of the API, and may also be obtained as part of the “Debugging Tools For Windows" package available as a free download from Microsoft, or as part of the Platform SDK. Also, with Windows' rules for finding DLLs (see above), this DLL **must** be placed either in the current directory or the directory that contains the mutatee executable on Windows versions earlier than XP. If you get an error message saying that SymEnumSymbols is not found in the DbgHelp.DLL, Windows has found an older version of the DbgHelp DLL. The depends.exe tool that ships with Visual C++ is a handy tool to determine which DLLs will be used when starting your mutator.

To locate procedure and variables in your mutatee, the API needs symbolic debug information. The API supports CodeView-format debug information, but currently requires the debug information to be included in the executable itself. (Recent Visual C++ versions place this information in a separate file by default.) Using Visual C++ 6, to include the debug information in the executable you should compile your code with the /Z7 option to generate CodeView symbols. You must also direct the linker to include the debug information in the executable using the /debug and /pdb:none linker options. Because the linker shipped with Visual C++ 7 does not support the /pdb:none option, currently Visual C++ 6 is required to build mutatees. Figure 3 shows a sample Makefile for the Microsoft Visual C++ compiler.

```
CC = cl /Z7

OBJECTS = foo.obj bar.obj

PDDIR = c:\paradyn\lib\i386-unknown-nt4.0

foo:  $(OBJECTS)
      link -out:foo.exe -debug –pdb:none \
      $(OBJECTS)
```

Figure 3: Sample Makefile for Windows NT.

The API needs to instrument some system libraries (in particular, kernel32.dll), and this can only be done if the symbols for the system libraries are installed. These symbols may be available from several sources depending on your version of Windows. For example, symbols for Windows NT libraries are available on the Windows installation disks. Also, Microsoft now provides an Internet-based symbol server to download symbols for system DLLs on demand.

6. EXAMPLES

This section provides a set of example code to show how the various parts of the API are used together.

6.1 Complete Example (retee)

In this section we show a complete program to demonstrate the use of the API. The example is a program called "re-tee." It takes three arguments: the pathname of an executable program, the process id of a running instance of the same program, and a file name. It adds code to the running program that copies to the named file all output that the program writes to its standard output file descriptor (so 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.

Using the API to directly create programs is possible, but somewhat tedious. We anticipate that most users of the API will be tool builders who will create higher level languages for specifying instrumentation. For example, the MDL language[4].

```
#include <stdio.h>
#include <fcntl.h>
#include "BPatch.h"
#include "BPatch_Vector.h"
#include "BPatch_thread.h"

BPatch bpatch;

main(int argc, char *argv[])
{
  int pid;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s prog_filename pid log_filename\n",argv[0]);
       exit(1);
  }

  pid = atoi(argv[2]);

  // Attach to the program
  BPatch_thread *appThread = bpatch.attachProcess(argv[1], pid);

  // Read the program's image and get an associated image object
  BPatch_image *appImage = appThread->getImage();

  BPatch_Vector<BPatch_function*> writeFuncs;
  appImage->findFunction("_write", writeFuncs);

  if(writeFuncs.size() == 0)
      return -1;

  // Find the entry point to the procedure "write"
  BPatch_Vector<BPatch_point *> *points =
  writeFuncs[0]->findPoint(BPatch_entry);

  if ((*points).size() == 0) {
      fprintf(stderr, "Unable to find entry point to \"write.\"\n");
      exit(1);
  }


  // Generate code that opens the file the first time it is called.

  // The code to be generate is:
  //  if (!flagVar) {
  //      fd = open(argv[3], O_WRONLY|O_CREAT, 0666);
  //      flagVar = 1;
  //  }
  //  (1) Find the open function
  BPatch_Vector<BPatch_function *>openFuncs;
  appImage->findFunction("open", openFuncs);

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

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

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

  //  (5) Push 3 && 4 onto the list from step 2
  openArgs.push_back(&fileName);
  openArgs.push_back(&fileFlags);
  openArgs.push_back(&fileMode);

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


  //  (7) allocate a variable to hold the open file descriptor
  BPatch_variableExpr *fdVar =
      appThread->malloc(*appImage->findType("int"));

  //  (8) create assignment statement of variable from step 7 to return
  //        value from step 6.
  BPatch_arithExpr openFile(BPatch_assign, *fdVar, openCall);

  //      (9) Find the integer type, and then allocate a variable
  //        of this type to be used as a flag to indicate if the
  //        open call was made on a previous call to write.
  BPatch_variableExpr *flagVar=
      appThread->malloc(*appImage->findType("int"));

  //  Declare a snippet vector to hold the list of items
  BPatch_Vector<BPatch_snippet *> initStatements;

  //  (10) flagVar = 1;
  BPatch_arithExpr setFlag(BPatch_assign, *flagVar, BPatch_constExpr(1));


  //  (11) make a sequence of the open and the assignment statements
  initStatements.push_back(&openFile);
  initStatements.push_back(&setFlag);
  BPatch_sequence initSequence(initStatements);

  //  (12) create expression (flagVar == 1)
  BPatch_boolExpr testFlag(BPatch_eq, *flagVar, BPatch_constExpr(0));

  //  (13) use expression #12 and statement #11 to produce if-statement
  BPatch_ifExpr initIfNeeded(testFlag, initSequence);


  // Generate the code that copies all writes to file descriptor 1
  // to our log file.
```

```
  // Call write with the same data but for our file descriptor
  // The C code we generate is:
  //     if (parameter[0] == 1) {
  //    write(fd, parameter[1], parameter[2])
  //        }


  // Find the write function call
  //BPatch_Vector<BPatch_function *>writeFuncs;
  //appImage->findFunction("write", writeFuncs);

  // Build up a parameter list with the items:
  //        1) The file description of our log file
  //  2) First parameter to the original function
  //  3) Second parameter to the original function
  BPatch_Vector<BPatch_snippet *> writeArgs;
  BPatch_paramExpr paramBuf(1);
  BPatch_paramExpr paramNbyte(2);
  writeArgs.push_back(fdVar);
  writeArgs.push_back(&paramBuf);
  writeArgs.push_back(&paramNbyte);

  // Create a function call snippet write(fd, parameter[1], parameter[2])
  BPatch_funcCallExpr writeCall(*writeFuncs[0], writeArgs);


  // (1) Build a vector of snippets with each statement being push on
  BPatch_Vector<BPatch_snippet *> copyWriteStatements;
  copyWriteStatements.push_back(&initIfNeeded);
  copyWriteStatements.push_back(&writeCall);

  //  (2) Convert the vector into a sequence
  BPatch_sequence copyWrite(copyWriteStatements);

  //  (3) Create the boolean expression ($param[0] == 1)
  BPatch_boolExpr compareFd(BPatch_eq, BPatch_paramExpr(0),
                      BPatch_constExpr(1));

  //  (4) Create if statement using expression from (3) and
  //        true clause from (2)
  BPatch_ifExpr logStdout(compareFd, copyWrite);

  // Insert the code into the thread.
  appThread->insertSnippet(logStdout, *points);

  //  continue execution of the mutatee
  appThread->continueExecution();

  // wait for mutatee to terminate and allow Dyninst to handle events
  while (!appThread->isTerminated())
            bpatch.waitForStatusChange();

  printf("Done.\n");
}
```

6.2 Instrumenting Memory Access

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

These snippets may be used to instrument a given instrumentation point if and only if the point has memory access information attached to it. In this release the only way to create instrumentation points that have memory access information attached to them is via BPatch_function.findPoint(const BPatch_Set<BPatch_opCode>&). For example, to instrument all the loads and stores in a function named foo with a call to another fuction bar that takes one argument (the effective address) one may write:

```
// assuming that thr points to some interesting thread
BPatch_thread* thr = ...;
BPatch_image *img = bpthr->getImage();

// build the set that describes the type of accesses we’re looking for
BPatch_Set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);

// scan the function foo and create instrumentation points
BPatch_Vector<BPatch_function*> fooFunctions;
img->findFunction(“foo”, fooFunctions);
BPatch_Vector<BPatch_point*>* r = fooFunctions[0]->findPoint(axs);

// create the printf function call snippet
BPatch_Vector<BPatch_snippet*> printfArgs;
BPatch_constExpr fmt("Access at: %d.\n");
printfArgs.push_back(&fmt);
BPatch_effectiveAddressExpr eae;
printfArgs.push_back(&eae);
BPatch_function *printfFunc = img->findFunction("printf");
BPatch_funcCallExpr printfCall(*printfFunc, printfArgs);

// insert the snippet at the instrumentation points
thr->insertSnippet(printfCall, *r);
```

APPENDIX A - RUNNING THE TEST CASES

This section describes how to run the dyninstAPI test cases. The primary purpose of the test cases is to verify that the API has been installed correctly (and for use in regression testing by the developers of the dyninst library). The code may also be of use to others since it provides a fairly complete example of how to call most of the API methods. The test suite consists of nine mutator programs (`test[1-9]`) and their associated mutatee programs. The mutatee programs are named `test`**`N`**`.mutatee_`**`COMP`** where **`N`** is the test number and **`COMP`** is the compiler used to build the mutatee (eg. `cc`, `gcc`, `g++`, `xlC`, etc.).

To compile the tests suite, type `make` in the appropriate platform specific directory under (…/dyninstAPI/tests). This should produce, depending on the platform, 8 to 24 programs and several shared libraries.

To run one of the tests, simply enter the test program name (e.g., test1). This will run the test, and the output should be a series of lines indicating each test number as it completes. In addition, the tests take the following command line arguments:

`-attach`

> Run the mutatee process and have the mutator attach to it rather than using the createProcess method. The -attach option is only available for test1 and test2.

`-relocate`

> Rewrite and relocate instrumented functions in mutatee instead of using the traditional trampoline based approach.

`-mutatee <mutatee name>`

> Run the mutatee named `<mutatee name>` rather than the default mutatee for this test. This is useful to run test cases with versions of the mutatee compiled with a systems native compiler in addition to the GNU compilers. If currently supported, the mutatee for the native compiler is named testN.mutatee_cc (see table at the end of this section for a list of platforms).

`-n32`

> Run the 32-bit version of the mutatee test. This flags is only valid on SGI platforms. This command line flag changes the shared libraries that are loaded to libtest?_n32.so, it also changes the mutatee to test?.mutatee_gcc_n32. If you want to test 32-bit mutatees compiled with the native compiler, use -n32 and -mutatee test?.mutatee_cc_n32. The order of -n32 and -mutatee is important.

`-run <subtest #> <subtest #> …`

> Only run the specific sub-tests listed. For example, to run sub-test case 4 of test2 you would enter `test2 -run 4`.

`-skip <subtest #> <subtest #> …`

Skip the specific sub-tests listed. For example, to skip sub-test case 4 of test2 you would enter `test2 -skip 4`. All other tests are run.

`-V`

Print out the name of the dynInst runtime library the will be used to run this test. This is useful to check that your environment is correctly setup to run mutator programs.

`-verbose`

Enable detailed debugging output. This is useful when trying to track down the reason that one (or more) of the test cases failed.

`-v+`

Enable the printing of warning level error messages (`BpatchWarning`) to standard output. This is useful for debugging the test cases.

`-v++`

Enable the printing of information and warning level messages via the error reporting callback function (`BPatchWarning` and `BPatchInfo`). These options are useful for debugging the test cases.

Some platforms do not implement all the test cases due to OS restrictions or missing OS features. If a test is not run on a specific platform, the message "`Skipped test #XX`" will be displayed. If any of the tests produces a line of the form "`**Failed test #XX`" there is something wrong with the version of the API or its installation. Each test should still produce a message of the form "`Passed test #XXX`", and a message at the end indicating that either all tests were passed, or all requested tests were passed (if the -run option is used).

Note: test2 produces a few lines that look like error messages since it is testing the error reporting features of the API (e.g. "file not found"). Check for the "All tests passed" message at the end to confirm correct execution.

The following table summarizes the current status of the various tests cases on different platforms and compilers. For each platform, the entry under the column for a test indicates any tests that are currently being skipped due to missing or unsupported features. The notes refer to other possible problems with the platforms. All platforms are built and tested with the gcc 3.3.3 compiler. Exceptions include MIPS and Windows, which rely on the native vendor compilers to build the Dyninst libraries.

Platform Information

PLATFORM	Mutatee Compiler(s)	Abbreviation
alpha-dec-osf5.1	gcc, native	osf5.1
i386-unknown-linux2.4	gcc	lnx2.4
ia64-unknown-linux2.4	gcc	lnx-64
i386-unknown-nt4.0	native	nt4.0
mips-sgi-irix6.4 (+)	gcc, native	irx6.5
rs6000-ibm-aix5.1	gcc, native	aix5.1
sparc-sun-solaris2.8 (32 bit)	gcc, native	sol2.8

Skipped Sub-Tests

	aix5.1	irx6.5	lnx2.4	lnx-64	nt4.0	osf5.1	sol2.8
Test 1	22, 35, 37	22-27, 30, 35, 37	37	37	21, 22, 35, 37	31, 35, 37	37
Test 2	9	9	9	9	6, 7, 9, 10, 13	7, 9, 13	
Test 3							
Test 4					N/A	N/A	
Test 5	1, 2, 4-8, 10, 11	1, 2, 4-8, 10, 11	11	6	N/A	1, 2, 4-8, 10, 11	
Test 6	4, 5, 7	4-8			N/A	4-8	
Test 7					N/A	N/A	
Test 8	2	2			N/A	N/A	3
Test 9	6	N/A		N/A	N/A	N/A	6
Test 10	N/A	N/A	N/A	N/A	N/A	N/A	
Test 11	N/A	N/A	N/A	N/A	N/A	N/A	

Notes

Platform	Note
i386-unknown-linux2.4	% Warnings generated: -attach "continue: No such process" test2 "wait returned status of an unknown process" We have tested with RedHat Version 9.0
mips-sgi-irix6.4	+ -n32 tests same as default (64-bit) case * Tests occasionally hang upon completion (sometimes freed by ps or telnet or ...)
i386-unknown-nt4.0	% Warnings generated: -attach "process::isRunning_() returning true" @ test4 not implemented on this platform
sparc-sun-solaris2.8	Test5 runs only with mutatees compiled using g++, not the native compiler.

APPENDIX B - COMMON PITFALLS

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

Attach followed by detach

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

Attaching to a program that has already been modified by dyninst

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

Memory access instrumentation on SPARC

Due to the peculiarities of the SPARC architecture adjacent loads and stores cannot be currently instrumented.

Thread support

Currently, the data structures used by DyninstAPI are not thread safe and DyninstAPI does not support threaded mutatees.

C++ exceptions

DyninstAPI does not currently support the usage of C++ exceptions in the mutatee.

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.

Stripped binaries

DyninstAPI cannot operate on mutatees that have been stripped (with GNU strip) of their symbols on AIX. On other platforms (Linux, Solaris) DyninstAPI can provide some subset of its functionality to a stripped mutatee but not all functionality.

64-bit binaries (Solaris & AIX)

DyninstAPI does not support 64-bit binaries on Solaris or AIX.

Missing or out-of-date DbgHelp DLL (Windows)

DyninstAPI 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 DyninstAPI. The binaries produced by the compiler do not contain the source file information necessary for DyninstAPI to assign the debug symbols to the correct module.

When Building DyninstAPI from Source

Commonly, required external libraries and headers (such as libdwarf or libelf) are not found correctly by the compiler. Often, such packages are installed, but outside of the compiler's default search path.

Because of this, we have provided the following extention to our make system. If the file make.config.local exists inside the $DYNINST_ROOT/core directory, it will automatically be included during the build process.

You can then set the makefile variables FIRST_INCLUDE and FIRST_LIBDIR inside make.config.local. These variables represent the compiler flags used during the compilation and linking phase, respectively. These paths will be searched before any others, insuring that the correct package is used. For example:

```
FIRST_INCLUDE=-I/usr/local/packages/libelf-0.8.5/include
FIRST_LIBDIR =-L/usr/local/packages/libelf-0.8.5/lib
```

INDEX

REFERENCES

1. B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications (to appear)*, 2000.
2. J. K. Hollingsworth and B. P. Miller, "Using Cost to Control Instrumentation Overhead," *Theoretical Computer Science*, **196**(1-2), 1998, pp. 241-258.
3. J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., pp. 841-850.
4. J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Nov. 1997, San Francisco, pp. 201-212.
5. J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *PLDI*. June 18-21, 1995, La Jolla, CA, ACM, pp. 291-300.