Backtracking and Dynamic Patching for Free

Bjorn De Sutter* Bruno De Bus Michiel Ronsse Koen De Bosschere
Electronics and Information Systems Department
Ghent University, member of the HiPEAC network
{brdsutte,bdebus,ronsse,kdb}@elis.ugent.be

ABSTRACT

We present a way to incorporate *backtracking* and *dynamic patching* into existing debuggers, without requiring any change to their source code, the compiler or the run-time environment. An implementation on top of **gdb** is presented.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Debugging aids

General Terms: Experimentation.

1. INTRODUCTION

Any software development project requires debugging. Some of this effort can be offloaded to automated tools, but other bugs need to be corrected through *cyclic debugging*.

Cyclic debugging is usually used to relate a bug's symptom to the bug itself. First, we often try to reduce the input data to the smallest set that still triggers a bug. In this phase, the debuggee is rerun multiple times. Thereafter, we try to locate the actual bug by inspecting the symptoms and additional program state, either by using a debugger, or by inserting additional print statements, assertions and consistency checks in the debuggee. Thus, we try to find the first point at which the program deviates from its expected behavior. This again can consist of multiple program runs. Usually, it is cumbersome to use a debugger while editing the debuggee. After each edit, it is necessary to recompile the debuggee and restart the debugger. This means that all breakpoints and watchpoints, the current program state, and all other information kept by the debugger is lost. Some commercial debugger features help to overcome this problem. DEC's `ladebug` [2] and Intel's `idb` [3] take checkpoints, at which the debuggee's state is saved. Later, the user can revert to this saved state. In the remainder of this paper we will call this *backtracking*. Another useful feature is the ability to patch a debuggee from within the debugger without recompiling the entire debuggee. Sun's `dbx` [5] and Apple's `xcode` [1] feature this dynamic patching. Unfortunately, no freely available debuggers offer both features.

*Supported by FWO-Vlaanderen.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AADEBUG'05, September 19–21, 2005, Monterey, California, USA.
Copyright 2005 ACM 1-59593-050-7/05/0009 ...$5.00.

This article shows that backtracking and dynamic patching can be added to an existing command-line debugger without the need to change that debugger, the development, or the run-time environment. For an example implementation, we used `gdb` (http://sources.redhat.com/gdb) that neither allows taking snapshots nor offers a mechanism to change the program at run-time. Similar tool-kits can be built on top of most `gdb`-like debuggers without requiring any insight in the internals of those debuggers however.

2. BACKTRACK DEBUGGING

A typical approach to checkpointing [6] is to fork a program at regular intervals. The forked copy dumps its internal state to disk while the original process continues to run. When the dumped state needs to be restored, it is loaded into main memory again, and as much as possible of the debuggee and environment state is restored. Thus, a checkpoint enables the return to a specific point in a program's execution. Although we could have implemented backtrack debugging on top of a checkpoint library, we found the whole checkpoint approach unnecessarily heavyweight. Checkpoints written to disk have the size of a core dump. Although parts of the program can be annotated in order to reduce the size of the checkpoint, a large amount of data needs to be stored, which slows down the execution.

2.1 Lightweight Checkpoints

In our lightweight approach, we simply duplicate the program state in memory by forking off a new process. Then the programmer can debug the child copy without disturbing the parent process, which waits for the child to die. This approach offers several advantages over regular checkpointing, but of course it also comes with its own limitations.

One advantage is that we never destroy connections with the run-time environment. Most often, we do not need to reopen files, sockets,... when we revert to a previously forked child. Unfortunately, this does not always hold. Properties of files and sockets, such as the current position in a file, are stored in the kernel and are thus shared. Problems arise when a forked child, e.g., reads from a file: As a side-effect the file position for the parent changes as well. Fortunately, this can easily be solved by saving the file position for all open files just prior to a fork, and resetting the file positions upon return to the parent process.

Additionally, simple forks come with an additional bonus on operating systems that implement copy-on-write. When a process is forked, all the memory pages allocated by this process are also duplicated *but not yet physically allocated.*

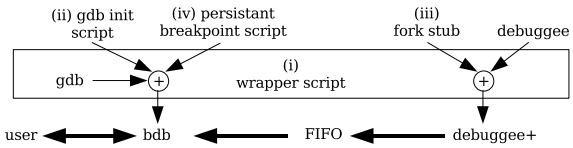

Figure 1: Overview of the checkpoint system.

```
1  #!/bin/bash
2  TMP=`mktemp -d "/tmp/bdbXXXXXX"`
3  gcc -g -rdynamic -shared -fPIC -Wl,-soname -DTMP_DIR_NAME=\"$TMP\"
       -Wl,fork_stub.so $TMP/fork_stub.c -o $TMP/fork_stub.so
4  gdb -x bdb_init --tty=`tty` $*
5  rm -fr $TMP
```

Figure 2: Backtrack debugging script

This allocation only happens when either the parent or the child modifies a page. As such, the operating system gives us incremental checkpointing for free.

Our simple forking approach is hampered by the same problems that regular checkpointing faces: It is difficult and sometimes even impossible to accurately restore the state of the run-time environment. Connections to other systems can get lost, file contents can be changed, it is impossible to un-send a sent network package, etc. To overcome this we would need techniques like input replay. For a large set of applications however, these problems do not occur.

2.2 Using `gdb` for Lightweight Checkpoints

To add the described lightweight checkpointing to a debugger, all we need to do is call `fork()` from within the debugger, make the parent wait for the child to die, detach from the parent and attach to the forked child. As debuggers usually allow to attach to a process, and to change the program counter and the memory and register contents, it is possible to "emulate" a call to `fork()`.

What remains to be done is bookkeeping: remember the process ID of the parent and add commands to the debugger to pick up the parent again. This can easily be done in the source code of the debugger, but we opted for an even simpler approach. We wrote a shared library that is in charge of the bookkeeping and that facilitates the communication with the debugger. This shared library is preloaded in the debuggee when it is loaded by the debugger, and its functionality is called from within the debugger.

Note that this is rather simple to implement. The implementation on top of `gdb` consists of four small parts: (i) a wrapper script around gdb to enable interprocess synchronization and communication, (ii) the C-source code that is used to call `fork()`, (iii) a gdb init script that defines the commands to take a checkpoint (`checkpoint`) and to go back to a checkpoint (`backtrace`), and (iv) a gdb script that enables the use of breakpoints in the presence of checkpoints. The cooperation between these parts is depicted in Fig. 1. The central part is (i) the wrapper script, that adds commands to gdb by means of the (ii) gdb init script and (iv) persistent breakpoints script, thus creating a Backtrack DeBugger *bdb*, with which the user will interact. The wrapper script also builds a new version debuggee+ of the debuggee, by linking (iii) the necessary fork code into it. This new version debuggee+ is executed under control of bdb, and communicates to bdb by means of a FIFO communication channel that is set up whenever a checkpoint is taken.

The wrapper script The central part of our checkpoint implementation, i.e., a shell script that wraps around gdb, is presented in Fig. 2. First, a directory is created that will hold temporary files and a named pipe (a FIFO) that will be used for interprocess synchronization. On line 3, two files are created: the initialization script for gdb (bdb_init) and a C-source file containing the code for forking and bookkeeping (fork_stub.c). The code to create them is not listed, but the resulting files are discussed in the next sections. The C-source file implementing the necessary forking is compiled on line 4. The result is a shared library that will be preloaded in the debuggee once the debuggee gets loaded by the debugger. This preloading is accomplished by the command on the first line of the gdb initialization script as depicted in Fig. 4. Note that we cannot include this line in the wrapper script, because if we did so, the forking code would also be preloaded into the debugger itself. On line 5, gdb is started, executing the commands in the initialization script. Afterwards, we delete all temporary files on line 6.

C-source for forking Fig. 3 lists the code to fork the debuggee and do the bookkeeping. For the sake of clarity, code to take multiple checkpoints is omitted. Lines 3–4 define two global variables that will be used to communicate information from the debugger to the forking code. The variable `bdb_tmp_dir` holds the name of the temporary directory in which the FIFOs will be created, and in which process numbers will be stored in files. The variable `wait_for_debugger` will be updated by the debugger (lines 13 and 21 in Fig. 4) to inform the forking code that the debugger has attached to a process, and that the process does no longer need to delay execution with the loop on line 25.

Lines 5–26 show the function that is called by the debugger (line 7 in Fig. 4) to take a checkpoint. Line 8 forks the debuggee program. Lines 9–23 are executed only by the parent process. In these lines, two simple gdb script files are created. The first is generated on lines 13–15, and is used by the debugger (line 12 in Fig. 4) to attach to the child process. The second is generated on lines 16–19, and is used to return to the parent process once a child has finished.

On lines 20–22, the new FIFO created in the gdb init script (lines 8–9) is opened, and immediately closed. The debugger, who starts listening to the FIFO on line 11 of Fig. 4 thus gets the signal that it can continue execution, after it had been blocked when it started listening to the FIFO. After this signal, the parent process waits in the loop on line 23, until his child dies. The seemingly endless loop on line 25 lets a debuggee continue when the debugger attaches to it. The child process enters the loop when a checkpoint is taken, the parent enters it when a child dies. The debugger sets the global variable `wait_for_debugger` to zero (lines 13, 21 in Fig. 4) when it wants a debuggee to continue.

GDB initialization script Fig. 4 depicts the script, written in gdb's script language, for initializing gdb. In the first line, the code to fork the program is preloaded into the debugger. Next, lines 2–16 define the command to take a checkpoint. This command is to be executed when the debuggee has reached a breakpoint that was previously set by the developer. Because this breakpoint must be removed from the debuggee before it is forked, all existing breakpoints are first deleted (line 3). Moreover, a fork cannot be executed while the debugger is attached to the debuggee. This is a gdb-specific problem. For this reason, we need to detach from the debuggee before the fork is executed, and reattach to the child once the fork has been finished. However, to call the forking code from within the debugger, the

```
1  #include <stdio.h>
2  #include <wait.h>
3  char bdb_tmp_dir[] = TMP_DIR_NAME;
4  volatile int wait_for_debugger=1;
5  void bdb_chckpnt_stub() {
6    int chckpnt_pid, parent_pid_pass=getpid();
7    wait_for_debugger=1;
8    if (chckpnt_pid=fork()) {
9      int stat;
10     char name[100];
11     FILE * fp;
12     sprintf(name,"%s/cont_from_chckpnt",bdb_tmp_dir);
13     fp=fopen(name,"w");
14     fprintf(fp,"attach %d\n",chckpnt_pid);
15     fclose(fp);
16     sprintf(name,"%s/backtrack_to_chckpnt",bdb_tmp_dir);
17     fp=fopen(name,"w");
18     fprintf(fp,"attach %d\n",getpid());
19     fclose(fp);
20     sprintf(name,"%s/fifo", bdb_tmp_dir);
21     fp=fopen(name,"w");
22     fclose(fp);
23     while(waitpid(chckpnt_pid,&stat,0)!=chckpnt_pid);
24   }
25   while(wait_for_debugger);
26 }
```

Figure 3: C-source to fork the program

debuggee still needs to be attached: The debuggee needs to be detached before the forking code is executed, but after the checkpoint script has called the forking code.

To implement this, we first set (on line 4) a new temporary breakpoint at the start of the code that will fork the program. Then, the current program counter is pushed on the stack (lines 5–6). This will be used as return address when the checkpointing procedure `bdb_chckpnt_stub()` returns in order to continue the execution of the debuggee once the fork has been executed. After that, control is transferred to the start of the checkpoint procedure (line 7), where the debuggee now starts executing. However, its execution immediately stops at the temporary breakpoint.

On lines 8–9, the FIFO (a pipe that blocks until something writes to it) is created. The debugger detaches from the process on line 10 to enable the fork. To detect when the fork is done, the debugger then starts listening to the FIFO until the child process is created in `bdb_chckpnt_stub()`. Once the debugger is notified through the FIFO, he picks up the forked child on line 12 with the `cont_from_chckpnt` script that was created by the checkpoint code. Concretely, the `attach` command is executed with the child process ID.

That child's execution is than triggered by assigning 0 to its variable `wait_for_debugger` on line 13 and issuing the continue statement on line 15. The command on line 14 to the reset persistent breakpoints, is discussed in Sec. 2.2.

Lines 17–23 list the gdb script code that implements the command to return to a checkpoint. The running child is killed (line 18). Then the `backtrack_to_chckpnt` script picks up the parent on line 19. Again, this script consist of the simple `attach` command with the appropriate process id. Finally, the endless while-loop in the parent is broken on line 21, and the parent resumes execution on line 22.

Persistent breakpoints Problems may arise when standard breakpoints are used. To add a breakpoint, a debugger replaces an instruction by an illegal instruction that causes a trap. The debugger remembers the original instruction to restore it when required. When the debugger detaches from a process however, it forgets the changes made, without first undoing them. When we later reattach to a process to continue its execution, and one of the changed instructions is reached, the debugger will not be able to restore the original instruction. To solve this problem, we simply delete all breakpoints when we checkpoint the program. To compensate for the lack of normal breakpoints, we implemented a

```
1  set environment LD_PRELOAD=$TMP/fork_stub.so
2  define checkpoint
3        delete
4        tbreak bdb_chckpnt_stub
5        set $sp=$sp - 4
6        set (*((int *) \$sp))=$pc
7        jump *bdb_chckpnt_stub
8        shell rm -f $TMP/fifo
9        shell mknod $TMP/fifo p
10       detach
11       shell cat $TMP/fifo
12       source $TMP/cont_from_chckpnt
13       set wait_for_debugger=0
14       source $TMP/pbreaks
15       continue
16 end
17 define backtrack
18       kill
19       source $TMP/backtrack_to_chckpnt
20       source $TMP/pbreaks
21       set wait_for_debugger=0
22       continue
23 end
```

Figure 4: Gdb initialization script

```
1  shell echo > $TMP/pbreaks
2  define pbreak
3       break \$arg0
4       shell echo break \$arg0 >> $TMP/pbreaks
5  end
```

Figure 5: Persistent breakpoints initialization

new kind of breakpoint, called *persistent breakpoints*, that are maintained in the shared bookkeeping library. To that extent, the code listed in Fig. 5 is added to the gdb initialization script. First, the list of persistent breakpoints is cleared. Lines 2–5 then implement persistent breakpoints by storing all breakpoints in a file. This script is invoked from within both the checkpointing and the backtracking script.

3. DYNAMIC PATCHING

Backtrack debugging as described so far requires running the debuggee from scratch after each modification: returning to a checkpoint implies going back to the old code. Because we often not only want to patch a debuggee to fix a bug, but also to insert more assertions or checks, we could save quite some execution time if we could dynamically patch the code, and rerun the patched code from some checkpoint. Patching a running program consists of three tasks.

Detecting changed parts First, we need to find out what part of the program has been changed since the start of the execution. Because replacing unchanged code by itself does not pose any problems, we opted for using simple time stamps. If the time stamp for a source code file is changed, we simply assume all code in this file is changed.

Loading revised code Secondly, we need to load patched code into a running program. Dynamic shared libraries make this easy. Just like we used scripting to load the bookkeeping library, we implement loading a dynamic patch library by means of a debugger initialization script.

Patching calls Finally, we need to replace calls to old procedures by calls to the revised versions in the dynamically loaded library. We opted for the automated approach of binary rewriting, in which a small binary rewriter replaces all procedures in a program by simple wrapper procedures that do nothing but jump to the original wrapped procedures via global procedure pointers. For dynamic patching, it then suffices to replace the pointers used in the wrappers.

This binary rewriting tool has to be run on the debuggee prior to its linking, to prepare it for dynamic patching before debugging. At link time, we have all the object files available that will make up the final program. These object files contain symbolic definitions of all procedures that are

exported from their modules (the global procedures). As we consider the whole object file revised as soon as one procedure in it is revised, it is only necessary to create wrappers for the exported procedures. Calls to local (non-exported) procedures will automatically reflect the revision as soon as calls to the exported procedures reflect their revision.

This simple approach has only one downside: a patch of some local procedure will only be in effect after the first call to an exported procedure of its module happens. When necessary, this can of course easily be solved by changing all static procedures in a program to global procedures with a unique name. Alternatively, one could provide wrappers for local procedures as well. When the programs are compiled with debugging information (which one would expect during the debugging of a program) all necessary information to do so is usually available in the debug information.

Upon rewriting the object files, we must consider two types of procedure calls: intermodular calls from a caller in one module to a callee in another module, and intramodular calls where caller and callee come from the same module.

For intermodular calls, the object file containing the caller contains a symbolic reference to the callee. The linker replaces this reference by the address of the callee during the relocation phase of the linking process [4]. This address is found in the object file of the callee, and more in particular in the defining symbol of the callee. In this case it suffices to replace the name of the callee in its defining symbol by a new, unique name. For example, callee `proc` becomes `wrapped_proc`. The wrapper procedure is created in a new object file, where its defining symbol now defines it as the original procedure (`proc`). The net result of this transformation is that after linking (and after the referencing symbol is resolved with the new defining symbol of the wrapper), any intermodular call instruction will not call the original procedure but the wrapper. Creating the wrapper procedures is very easy, and can be done by creating a short assembly fragment for each external procedure. These fragments are assembled and linked into the executable.

The second case, of intramodular calls, is more tricky, as there is only one symbol involved: the defining symbol of the callee is now also the symbol referenced by the call-sites. Simply changing the name of the callee in this symbol would not have the desired effect, since all call-sites would still point to the same procedure, albeit through the new name. So instead of renaming the callee, we add a new defining symbol for it, defining it with the new name (`wrapped_proc`). The original symbol is transformed into a so called *weak symbol* [4] with the original name `proc`. Weak symbols are special symbols that are only used during the linking process when no other symbols with the same name are found. As we also add a wrapper procedure with the name `proc` to the program, calls to `proc` through the weak symbol will be diverted by the linker to call the wrapper procedure that is defined by a normal symbol with the same name `proc`.

To implement the loading of revised code and patching of calls, we wrote a small tool that produces a shared library from a set of (revised) source code files. This library is loaded into the programs. It contains the code to replace the global procedure pointers of the revised and wrapped procedures that are used by the wrappers introduced in the next section. After the shared library is loaded into the debuggee, the replacement code will be called and from then on all calls will indirectly call the revised procedures.

```
1  #!/bin/bash
2  ...
3  for FILE in $INFILES; do
4  EXTERN_FUNS_PREFIX='^[0-9a-f]* g *F [^ ]* *[0-9a-f]* '
5  FUNS=`objdump --syms $FILE | sed -n "s/$EXTERN_FUNS_PREFIX//gp"`
6  echo $FUNS | tr " " "\n" | sed "s/^\(.*\)$/$CODE/g" | tr "#" "\n" >> $TMPDIR/add.s
7  ...
8  OBJCOPYPARAMS=`echo $FUNS | tr ' ' '\n' | sed "s/^\(.*\)$/--weaken-symbol \1/g"`
9  objcopy $OBJCOPYPARAMS $FILE $TMPDIR/weak_$DIRFREENAME
10 LDPARAMS=`echo $FUNS | tr ' ' '\n' | sed "s/^\(.*\)\$/--defsym $UNIQUE_NAME\1=\1/g"`
11 ld -r $LDPARAMS $TMPDIR/weak_$DIRFREENAME -o $TMPDIR/$DIRFREENAME
12 rm \$TMPDIR/weak_$DIRFREENAME
13 done
14 if [ \$PATCH ]
15 then
16 gcc -rdynamic -shared -fPIC -Wl,-soname -Wl,$OUTPUT $TMPDIR/add.s $FILE2 -o $OFILE
17 else
18 cc  -rdynamic -o $OUTPUT $TMPDIR/add.s $TMPDIR/change_stub.c $FILE2 -ldl
19 fi
```

Figure 6: The core of the object rewriter

To generate the "patchable" executable and the shared library for a patch, it suffices to change the build rules inside the `Makefile` of the debuggee: to generate the executable we simply need to execute the rewriter instead of the linker, and to create the shared library we execute the library building tool that creates the patch for all changed source code files. Which files have changed is detected by the `make` tool.

3.1 Implementation

While the application of binary rewriting techniques might seem complex, it is in fact based on scripts that invoke open source GNU tools such as `objcopy`, `ld`, `tr`, and `sed`.

Binary rewriter Fig. 6 lists the binary rewriter. The code for parameter parsing and for some utility functions are not listed for simplicity. The original input files (to the debuggee) are processed one by one (line 3). The name of the current object file is kept in the variable `$FILE`. Lines 4–5 query the object file to get the name of all functions in it, and store these names in the variable `$FUNS`. For each function in the object file we create some code, which is held in `$CODE`. This code depends on whether we are rewriting the original binary (code in Fig. 7) or creating a patch (code in Fig. 8), and it is selected in the omitted parameter parsing code. On line 7 a unique temporary name is generated for the rewritten object file, and stored in `$DIRFREENAME`.

In lines 8–11, we will rename all exported functions, and replace all references to the renamed functions with references to their wrappers. Although one can rename functions directly with the tool `objcopy`, doing so would also change all references (by name) to those renamed functions. We need to avoid this, because all references need to refer to the wrappers in the rewritten code instead. Therefore, we require the slightly more complex approach through weakening, which is implemented on lines 8–11.

On lines 8–9, an original object file is transformed into a new object file in which all functions are weakened. This means that the symbols defining the functions become weak symbols [4]. Weak symbols are overridden by other symbols during linking. As we define wrapper functions with the original procedure names, the effect of this weakening will be that the weakened symbols are ignored when we relink the debuggee. Instead, all symbolic references will now refer to the wrappers. Consequently, all calls to these weakened functions will now call the wrappers. Of course, the wrappers themselves still need to call the weakened functions. Lines 10–11 create a unique new, non-weak symbol for each of them, to which the wrapped calls then refer.

Finally, either the shared library that makes up a patch is created on line 16 in case a patched version needed to be made, or the rewritten debuggee is linked, in case we

```
1  CODE="                                          \
2  .data                                          #\
3  .align 4                                       #\
4  .globl address_of_$UNIQUE_NAME\1               #\
5  .type  address_of_$UNIQUE_NAME\1,@object       #\
6  .size  address_of_$UNIQUE_NAME\1,4             #\
7  address_of_$UNIQUE_NAME\1:                     #\
8  .long  $UNIQUE_NAME\1                          #\
9  .text                                          #\
10 .align 16                                      #\
11 .globl \1                                      #\
12 .type  \1,@function                            #\
13 \1:                                            #\
14 jmp    *address_of_$UNIQUE_NAME\1              #"
```

Figure 7: The code for the stubs

```
1  CODE="                                                  \
2  movl address_of_$UNIQUE_NAME\1@GOT(%ebx), %edx         #\
3  movl $PATCH_NAME\1@GOT(%ebx), %eax                      #\
4  movl %eax, (%edx)                                       #"
```

Figure 8: The code for changing function pointers

want to start debugging the original debuggee. In case the debuggee is linked, we also include a piece of C-code that will be used by the debugger to apply a patch. This piece of code is described in Sec. 3.2.

Code added to the debuggee When preparing the original debuggee for dynamic patching, wrappers are added for all exported functions. Fig. 7 shows the code of these wrappers on an x86-linux system. It consist of a function pointer definition (lines 1–8) that will be used to call the original function from within the wrapper, and of the small wrapper function itself (lines 9–14). The `\1` in the code is replaced by the name of the original function and `$UNIQUE_NAME` is used to make all variable names unique.

Code added when we create a patch When we create a patch, we add one instance of the code fragment of Fig. 8 to the program per revised function. Together, these instances form one new function, that will be called from within the debugger. For each revised function, the corresponding instance of the code fragment replaces the function pointer used by the function's wrapper by its new value, thus redirecting all future calls to the revised version.

Code called by the debugger to apply a patch Similar to the compiled C-code that was added to a debuggee to enable forking, we add compiled C-code to implement patching. This time, the C-code in Fig. 9 is called by the debugger when the user commands a dynamic patch with the command `call do_switch`. Before this call, the name of the patch (i.e., the dynamic library) needs to be assigned to the variable `patchname`. The shared library is opened on line 6. On line 13, the address of the function that changes all function pointers is obtained from the shared library. This function is called on line 19, which applies the patch. After this the function `do_switch` returns and normal program execution continues under the control of the debugger.

3.2 Reliable Dynamic Patching

With dynamic patching the programmer can stop and patch a program at any time. When the only change of the patch is to provide additional information about the program state, this poses no problems. When a patch changes the program behavior however to fix a bug, it is obvious that this patching process involves some hazards.

Mixing the use of two different versions of a procedure can cause a program to behave incorrectly, even if both versions are correct on their own. The versions might, e.g., consume or modify global data in a different manner, creating an inconsistent state when one version is called after the other.

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3  #include <sys/ptrace.h>
4  char patchname[1000]="\0";
5  void do_switch() {
6     void * patches=dlopen(patchname,RTLD_GLOBAL | RTLD_NOW);
7     void (*sym)();
8     if (!patches) {
9        printf("Could not find patch named %s\n",patchname);
10       printf("DLERROR: %s\n",dlerror());
11       return;
12    }
13    sym=(void (*)()) dlsym(patches,"init_change");
14    if (!sym) {
15       printf("Init change not found\n");
16       return;
17    }
18    printf("Patch %s is OK\n",patchname);
19    sym();
20    printf("Patch applied\n");
21 }
```

Figure 9: C-Source to apply the patch

Obviously, it is always safe to backtrack to the first execution of the first patched procedure. This can be implemented by automatically taking checkpoints whenever a procedure is first called. When a patch is applied, the debugger first automatically rewinds to the correct position. To limit the overhead of taking checkpoints upon entry to uninteresting procedures, the user can specify which procedures are candidates for revisions. Furthermore, if a patch does not affect the execution of the patched procedures before the execution of the bug, it is sufficient to backtrack to a checkpoint taken before the bug, and to continue there.

4. USE CASE

The described tool-set is ideally suited to debug long-running applications that operate on large data sets in phases. For example, we used the tool-set to debug a link-time optimizer, in which all IO happens at the beginning (reading in a program) or at the end (writing the optimized program). In between, an interprocedural control flow graph is transformed in a number of minute-long phases. To avoid having to execute all earlier phases when a bug in a later phase was discovered, backtracking and dynamic patching was often used. For example, after a fork had taken place, the user of the optimizer was asked for a range of basic blocks (out of hundreds of thousands) on which the buggy transformation was to be applied. Thus, a binary search for the basic block on which the buggy transformation was went wrong could often locate a bug in seconds instead of hours or days.

5. CONCLUSIONS

We presented backtracking and dynamic patching implementations that require little effort, showing that existing operating systems and debugging tools readily provide much of the required functionality.

6. REFERENCES

[1] Apple Computer. Introduction to fix and continue. http://developer.apple.com/documentation /DeveloperTools/Conceptual/PatchCommand/.

[2] Compaq Computer Corporation. Ladebug debugger manual. http://h30097.www3.hp.com/docs/base_doc/ DOCUMENTATION/V51A_HTML/LADEBUG/TITLE.HTM.

[3] Intel Corporation. Intel(R) debugger (IDB) manual.

[4] J.R. Levine. *Linkers and Loaders*. Morgan Kaufman, 2000.

[5] Sun Microsystems. Debugging a program with dbx. http://docs.sun.com/app/docs/doc/805-4948, 1999.

[6] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proc. Usenix Winter 1995 Technical Conference*, pp. 213–223, 1995.