Efficient Algorithms for Bidirectional Debugging

Bob Boothe
Computer Science Dept.
University of Southern Maine
Portland, ME 04104-9300
boothe@cs.usm.maine.edu

Abstract

This paper discusses our research into algorithms for creating an efficient bidirectional debugger in which all traditional forward movement commands can be performed with equal ease in the reverse direction. We expect that adding these backwards movement capabilities to a debugger will greatly increase its efficacy as a programming tool.

The efficiency of our methods arises from our use of event counters that are embedded into the program being debugged. These counters are used to precisely identify the desired target event on the fly as the target program executes. This is in contrast to traditional debuggers that may trap back to the debugger many times for some movements. For reverse movements we re-execute the program (possibly using two passes) to identify and stop at the desired earlier point. Our counter based techniques are essential for these reverse movements because they allow us to efficiently execute through the millions of events encountered during re-execution.

Two other important components of this debugger are its I/O logging and checkpointing. We log and later replay the results of system calls to ensure deterministic re-execution, and we use checkpointing to bound the amount of re-execution used for reverse movements. Short movements generally appear instantaneous, and the time for longer movements is usually bounded within a small constant factor of the temporal distance moved back.

1 Introduction

The purpose of a program debugger is to assist the user in locating programming errors. Standard capabilities allow stopping a program in mid-execution, deliberately and precisely moving along its execution path, and examining the state of the program, such as its variables and stack. A traditional *forward debugger*, such as dbx or gdb can move only in the direction of forward execution. The user specifies an amount to move forward, usually to the next statement or to the next breakpoint, and the debugger executes forward until it reaches that point. Forward movement is natural and well understood, however it is often exactly the opposite of what would be most convenient for a user trying to track down the cause of an error.

In this paper we discuss our research into algorithms for creating an efficient bidirectional debugger in which all traditional forward movement commands can be performed with equal ease in the reverse direction. We have named our prototype bidirectional debugger **bdb**.

Adding these backwards movement capabilities to a debugger will greatly increase its efficacy as a programming tool. Bugs become evident to the user when erroneous output is produced or when an error detection routine notices something wrong. Rather than start at the beginning of the program's execution, far removed from the location of the bug, the programmer can now start at the point where the bug manifests itself. From this point they can directly chase down in reverse how the program got there and where incorrect values originated.

In contrast, current debugging practice often involves a frustrating process of trying to *"sneak up on a bug"*. In a forward debugger, if we wish to arrive at a point just prior to where our bug occurs, we have no choice but to start at the beginning of the program. In stepping through the program, we often must guess if a certain function is worth examining or if we should step over it. If we set a breakpoint we often must guess how many breakpoints to continue past until we will arrive at the one of interest. If we ever make a single misjudgment and step past the bug, we must start over at the beginning of the program and try again. For bugs that occur deep into the program, we must step boldly forward or risk spending an eternity inching our way through the program. However the more boldly we step forward, the more we risk overstepping the bug. With a bidirectional debugger it will become trivial to undo a *"stepped past the bug"* mistake. Moreover, with an efficient bidirectional debugger it will be far easier and faster to simply start at the manifestation of the bug and work one's way backwards.

Bdb works with the C and C++ languages running on Digital/Compaq Alpha based UNIX workstations, but the techniques developed in this research are widely applicable.

1.1 Brief Background & Overview

There have been two general approaches for building bidirectional debuggers: "history logging" and "re-execution". The history logging approach creates a log of the values taken on by every variable as the program executes. Its problems are: (1) the rapid growth of the history log and (2) the

PLDI 2000, Vancouver, British Columbia, Canada.

overhead of collecting and saving so much information. The re-execution approach instead transparently re-executes the program to the desired earlier point. Its difficulties are: (1) locating the desired earlier point while re-executing forward, (2) the overhead involved in re-executing, and (3) ensuring deterministic re-execution. We will discuss the related research in more detail at the end of this paper after we have explained our own work.

In our research we have used the re-execution approach. We address issue 1 (locating earlier points) by embedding counters into the program being debugged, and stopping only when the desired counts are reached. We address issue 2 (overhead of re-execution) by using true re-execution rather than emulation. We address it by the low overhead of our embedded counters, and we address it by using periodic checkpoints to limit the extent of re-execution to a limited portion of the total execution. We address issues 3 (deterministic re-execution) by capturing and logging the original return values of system calls and then later replaying those values during re-execution.

1.2 Efficiency

The movement algorithms that we describe in this paper may not at first glance appear to be particularly efficient; we insert voluminous calls to counter routines, we re-execute the program even to move short distances backwards, and in some cases we will even perform multiple re-execution passes. However, the counter calls are simple and have only moderate overhead, and we use checkpointing to limit the amount of re-execution performed. In the final analysis of the whole system, all short forward or reverse movements will appear instantaneous to a human operator, and longer movements going far into the future or far back into the past will complete within small constant factors of the temporal distance moved. While the computer is doing a lot of work on the user's behalf, on the human time scale this debugger appears to works naturally and quite efficiently.

1.3 Shortcomings of Trap Based Debugging

A traditional debugger, such as gdb, dynamically inserts trap instructions at anticipated stopping points. For example, for a "next" command it inserts a trap at the start of the next statement. For a "breakpoint" it inserts a trap at the target statement, and for a "finish" command it inserts a trap at the return address of the current function call. The debugger then executes the program until it hits the trap statement.

This generally works well but can suffer serious performance degradation in cases where the debugger performs a large number of trap/resume cycles before it arrives at the desired point. An obvious example occurs when the user issues a counted command such as "continue 10000". This might be used, for example, to get to the last iteration of a loop. In this case the debugger will go through 10000 trap/resume cycles. Gdb incurs an overhead of approximately one million processor cycles for each trap/resume cycle due to the cost of context switching and the system calls used to evaluate the state of the program after each trap.

A less obvious situation occurs when the user tries to "next" over a recursive call. The statement that the trap is placed at may be encountered thousands or even millions of times before the recursive call is completed, and each of these incurs a trap/resume cycle. A similar situation occurs when trying to finish out of a non top-level recursive call.

```
step_cntr()
{
  step_cnt++;
  if (step_cnt == sc_stop_val)
    Trap to the debugger;
}

func_entry()
{
  call_depth++;
}

func_exit()
{
  call_depth--;
}
```

Figure 1: The basic step counter, function entry, and function exit routines.

We are also aware of other performance anomalies and bugs in trapped based debuggers, but these do not appear to be as universal as the aforementioned problems. All of these problems have been eliminated by the new techniques used in bdb.

2 Counter Based Movements

We have abandoned the traditional debugger implementation technique of inserting trap instructions at potential stopping points in the program being debugged. In its place we have developed techniques which use a collection of embedded counter routines to track the progress of the program and stop it precisely at the final target location. While these embedded counters add some overhead, they allow us to efficiently and precisely move backwards to earlier points in the execution by re-executing the program and stopping at earlier counter values.

We have implemented the forward movements: "step", "continue", "next", and "finish", and the analogous backwards movements which we have named: "bstep", "bcontinue", "previous", and "before". All of these commands can be given a count argument to go to the n^{th} occurrence. We have also implemented "until" and "buntil" movements to find where variables change, and we provide a general undo movement.

Our basic counters are a "step counter" and a "call depth counter". When the program is compiled for debugging, we insert calls to the step counter at the traditional debugger stepping points (each line starting a new statement). We also insert at each function entry and exit point a call to increment or decrement the call depth counter. Pseudo code for these counters is shown in Figure 1. (The actual code is the same except that the function names are prefixed with "bdb_" to reduce the possibility of name conflicts, and the variables are actually members of a global structure.)

The step counter provides an underlying time line for measuring the execution of the program being debugged. It is used by all of the movement commands. The call depth counter is used by the more complex movements such as finishing the current function invocation. Inserting these calls at compile time establishes call sites for calls to the counter routines. For many of the movements we use specialized counter routines. These specialized counters are inserted dy-

namically either by replacing the call at an individual call site or by changing an indirect jump to redirect an entire class of counter calls.

2.1 Step & Bstep

The *step* movement is the simplest debugger movement. It steps to the next source level statement in the program, stepping into function calls. Likewise *bstep* steps to the preceding statement, possibly stepping back into a preceding function call.

These movements are implemented simply and efficiently using the step counter. This counter increments a global step count and compares it to a stopping value. When the stopping value is reached the counter routine executes a trap which transfers control back to the debugger. This is efficient even for huge values of n because the user program only traps back to the debugger once. For typical user programs running on a 600 MHz processor, we see about 20 million steps per second, with an overhead of less than a factor 2. (We will provide more comprehensive performance measurements in Section 4.) In contrast, a traditional debugger bogs down unbearably for large counts because it traps at each step as described in Section 1.2.

We use 64-bit counters, which are vastly more than adequate for any conceivable program execution today. However if computer performance continues to improve at its current exponential rate, in 30 years we will need to consider the possibility of counters rolling over or consider using larger counters.

The real benefit of the step counter, however, is for stepping backwards. Suppose the user has run their program for 1 second and stopped where the step count is 20 million, to *bstep* one step we set the stop value to 1 less than the current count and re-execute the program from the beginning. With an efficient step counter this takes only 1 second since it only traps to the debugger when it is done. (In Section 5 we will discuss our use of checkpointing to bound the extent of re-execution proportional to the distance moved back.)

The points at which user issued movement commands will stop when using bdb are the set of *step* points in the program. This has a side benefit of providing a simple mechanism for controlling what body of code is of interest to the user. Most traditional debuggers have the often undesirable behavior of stepping into library routines. In most cases, especially for students, this is annoying and frustrating. This doesn't happen for bdb because the library functions were not compiled for debugging and thus do not contain any calls to bdb counter routines. Library routines are thus stepped over as if they were basic atomic operations. A sophisticated user, working on a large software project, might not want the debugger to step into certain trusted software modules (such as the constructor calls and other member functions of a class). They can achieve this by simply not compiling those files for debugging.

2.2 Continue & Bcontinue

"*Continue* n" means run forward until we reach the n^{th} statement with a breakpoint. Analogously "*bcontinue* n" to the user means run backwards until we reach the n^{th} previous statement with a breakpoint. Of course we can't really run backwards, but instead we locate the desired point while re-executing forward.

Figure 2 shows a simple diagram that we will use to help in explaining many of our movement algorithms. The figure shows an execution time line moving from left to right.

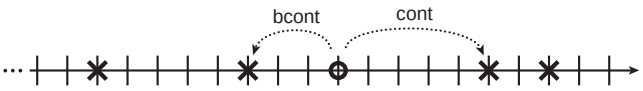


Figure 2: Execution time line showing "continuing" either forward or backward to the closest breakpoint.

```
brkpt_cntr()
{
  step_cnt++;
  brkpt_cnt++;
  if (brkpt_cnt == bc_stop_val
      || step_cnt == sc_stop_val)
    Trap to the debugger;
}
```

Figure 3: The breakpoint counter function.

The tick marks represent the basic *step* points. The circle represents the current position of the debugger along the execution path, and the X's represent steps points at which breakpoints occur. Figure 2 thus shows what will happen if the user issues either a *continue* or *bcontinue* command: the debugger will either move forward or backward to the next breakpoint. This is an abstract view of program execution that simply focuses on the execution time line. Language constructs such as loops would just appear as long sequence of step points. The multiple breakpoints shown on this diagram might in fact represent a single breakpoint placed in the program that is encountered repeatedly for each iteration of a loop.

The set of *step* point locations in a program occur at fixed locations determined at compile time, but breakpoints are inserted and removed at run time. Where a traditional debugger inserts a trap instruction, we insert a call to the breakpoint counter instead. This call replaces the normal call to the step counter, and thus the breakpoint counter replicates the work of the displaced software counter call as well as incrementing and testing the breakpoint count. This counter routine is shown in Figure 3. The performance advantage of using counter routines over the traditional trap instructions is crucial during re-execution when we may have to proceed past thousands of breakpoint occurrences before we arrive at the one of interest to the user.

Since breakpoints are inserted and deleted throughout a debugging session, we don't have a single reliable time line like we do with the step counter. Any change to the set of breakpoints invalidates our accumulated breakpoint counts. This doesn't impact our ability to perform forward movements because we just care about the relative change in the breakpoint count when executing forward. However for backwards movements, when the set of breakpoints changes, our task becomes harder. Our implementation uses two re-execution passes. The first pass establishes a valid breakpoint count up to the current position. The second pass then executes to the appropriate breakpoint count for the desired number of breakpoints moved back. Once breakpoint counts have been re-established in this fashion, future *bcontinue*'s require just one pass.

For most of our backwards movements there will be situations in which two passes are required. Although we have also investigated possible one pass algorithms for most of these movements, when the overall performance is consid-

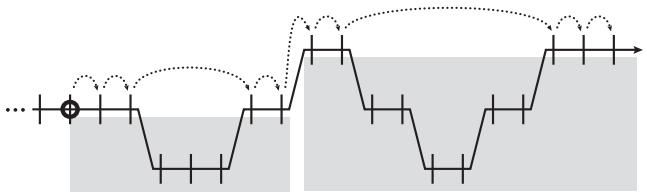


Figure 4: Sequence of points visited by *next*. Shading represents the base depth level below which stepping points would not be stopped at by *next*.

ered in Section 4 we will conclude that the two pass algorithms are preferable. Furthermore, when checkpoint is added in Section 5, the time spent on the second passes will be mostly eliminated anyways.

2.3 Next

Next and *previous* are the most challenging and interesting of the movements to implement. They have their own specialized counter routines for counting occurrences of the desired event. The *next* counter, for example, counts the points that would be stopped at by a series of *next* commands. These specialized counters replace the standard step counter. Since the calls to the step counters throughout the program are in fact indirect calls, we can efficiently change an entire class of counter calls by modifying just a single jump instruction.

Next moves to the next source line; it skips over function calls, and it steps out of functions when they return. To implement this we need a method to know when we step into and return from calls so that we can suspend counting of *next* points appropriately. The call depth counters on function entry and exit introduced at the start of Section 2 were invented for this purpose. The call depth is tracked for all movement commands so that we will have its correct value when it is needed for commands such as *next*.

The algorithm used for counting *next* points will be explained with the aid of Figure 4. In this figure, a change in level down represents a call to a function and thus an increase in call depth, and a change in level up represents a function return and a decrease in call depth. Figure 4 thus shows what will happen as the user issues a series of *next* commands or a counted *next* command.

To perform a *next* operation, the debugger starts by setting a variable named "base_depth" to the current call depth. The base depth is represented by shading in the figure. *Step* points occuring below the base depth are not counted as *next* points, and thus *next* will correctly skip over function calls. When a *next* operation returns up from the base depth, as shown in the diagram by the 5th arrow, the base depth is raised so that future counting will reflect the new base depth of further *next* operations.

The pseudocode for the counter routines used by *next* are shown in Figure 5. As described so far, the testing of the *step* counter doesn't make sense because we are looking for *next* points rather than *step* points. However with checkpointing (described in Section 5) the *step* counter will be used by the debugger to regain control at checkpoint boundaries. There is also a second version of the *next* counter routine (not shown) that increments and tests the breakpoint counter in addition to what is shown here. This second version is called from locations containing breakpoints, and it allows stopping a *next* movement prematurely when it encounters

```
next_cntr()
{
  step_cnt++;
  if (call_depth == base_depth)
    next_cnt++
  if (next_cnt == nc_stop_val
      || step_cnt == sc_stop_val)
    Trap to the debugger;
}

next_func_exit()
{
  call_depth--;
  if (call_depth < base_depth)
    base_depth--;
}
```

Figure 5: Specialized versions of counter routines for *next*.

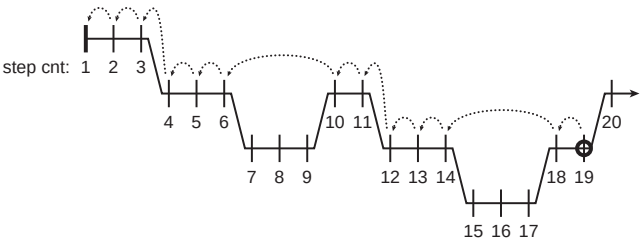


Figure 6: Sequence of points visited by *previous*.

a breakpoint before reaching its target destination.

2.4 Previous

Previous is our backwards analogue of *next*. It moves to the previous line; it steps over function calls and back up and out of a function entry. This is illustrated in Figure 6. *Next* and *previous* are not actually inverses. For example in Figure 6, a *next* from the current point would advance to step 20, but a *previous* from step 20 would skip over the preceding function call all the way back to step 11.

Although *previous* is conceptually similar to *next* in that it doesn't step below the base depth and it steps up out of functions (just in reverse), it is more difficult to implement because we can't really execute in reverse, but instead we must figure out the correct stopping point while re-executing forward! In designing this algorithm, we considered several different approaches. We present here the algorithm that was chosen as the best compromise based on speed, space, and the expected common cases for usage.

We use a two pass algorithm which we explain using Figure 7. The first pass gathers information. It starts by counting every *step* point as a potential *previous* point. When it steps into a function call (while moving forward), it continues counting *previous* points since the function just stepped into might turnout to be a function that would be stepped out of in reverse by *previous*. When we return from a function call (while moving forward), we learn that the function just counted would thus not have been visited by a *previous* command since *previous* would never step down into a function in reverse. This situation is shown in Figure 7: (while moving forward) the program steps into a function when going from *step* 6 to *step* 7, and it subsequently steps

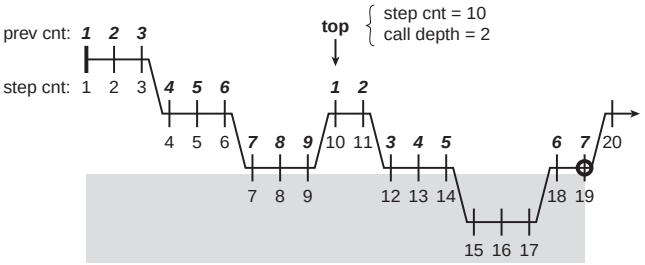


Figure 7: *Previous* counts calculated on first pass.

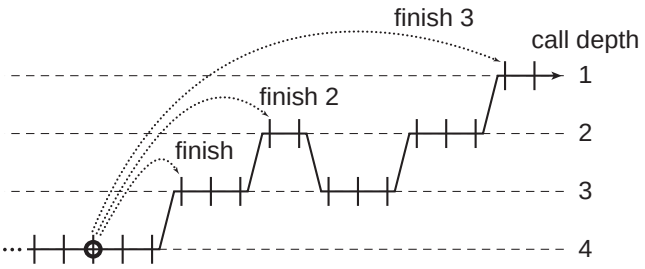


Figure 9: Destinations of *finish* commands.

```
prev_cntr()
{
  step_cnt++;
  if (call_depth <= base_depth)
    prev_cnt++
  if (prev_cnt == pc_stop_val
      || step_cnt == sc_stop_val)
    Trap to the debugger;
}

prev_func_exit()
{
  call_depth--;
  if (call_depth < base_depth)
    {
      prev_top_sc = step_cnt + 1;
      prev_top_depth = call_depth;
      prev_cnt = 0;
    }
}
```

Figure 8: Specialized counter routines for *previous*.

out of that function when going from *step* 9 to *step* 10. At this point, any *previous* points that were counted in that function are invalid (those at *step* points 7, 8, and 9 in the example). In our chosen algorithm we simply abandon the current count and start afresh.

We have labeled the new starting point as "top". This will be the top of the upcoming counted sequence of descending steps. Our counter routines save away the values of the *step* count and the call depth at this point. These will be used in the second pass to help navigate to the desired stopping point.

As we continue counting, if we step into functions below the base depth of the starting position of this command (as shown by shading in Figure 7), we stop counting *previous* points until we return up to the base depth. We do not have to reset "top" in this case since we have not miscounted any *previous* points.

After the first pass the debugger considers the gathered information. In the example, we are at the 7th *previous* point subsequent to "top". If the user had asked to *previous* from 1 to 6 *previous* points, we proceed to the second pass and re-execute to that point. In the unlikely event that the user asked to *previous* further back than the top point, the debugger performs additional "first" passes gathering *previous* information prior to the top point.

The counter routines for the first pass are shown in Figure 8. The second pass uses the normal step counter to reach the top point, and from there it uses the *previous* counter to reach the desired *previous* point.

We chose this algorithm because it works well for the expected common case usage of *previous*. A typical user will either move to the previous statement, or possibly use a small count value to move to somewhere that they could predict the destination. This would likely be inside the same function invocation or just backing out of it. Since in two passes we can reach any *previous* point within the same function invocation or arbitrarily far back along a descending chain of calls, we handle the common cases in two passes. (There is one subtle situation discussed in Section 3 that takes 3 passes to reach the calling function.)

We exploit the fact that a typical user will issue a series of *previous* commands. Repeated commands take only 1 pass because the relevant counters have already been gathered and saved for future use. Even in the case where we *previous* back to the top point, a subsequent *previous* will still take only 1 pass because for that case we gather new *previous* counts during re-execution to the top point in preparation for the possible repeated command.

Just as a *next* movement will be interrupted by the first intervening breakpoint, so should a *previous* movement. We wish to preserve the user's illusion that the debugger is executing backwards, so this breakpoint would in fact occur chronologically after we reach the desired *previous* point while re-executing forward. Bdb handles this situation by noting beforehand how many preceding breakpoints are expected, and if they have not all been passed by the time we reach the target point, bdb advances the program forward until it hits the last of these breakpoints. Just as for *next*, there are specialized versions of the *previous* counters that are used at breakpoint locations.

2.5 Finish & Before

The *finish* command continues until it finishes execution of the current function. A *finish n* command finishes *n* levels of nested calls. This is shown in Figure 9.

Finish can be implemented straightforwardly by testing the call depth. The debugger installs the *finish* counter routine in place of the *step* counter in the child, sets the stopping point to the desired call depth, and resumes the child. At each *step* point the *finish* counter checks to see if the call depth has reached the desired depth and traps back to the debugger when it is reached.

Before is once again similar, but more difficult than its analogous forward movement because we must identify the correct stopping point while executing forward. Figure 10 shows the destinations of example *before* commands.

In our chosen implementation we again use a two pass algorithm. The first pass determines the *step* count corresponding to the desired *before* point. During the first re-execution pass the *before* counter saves the *step* count when-

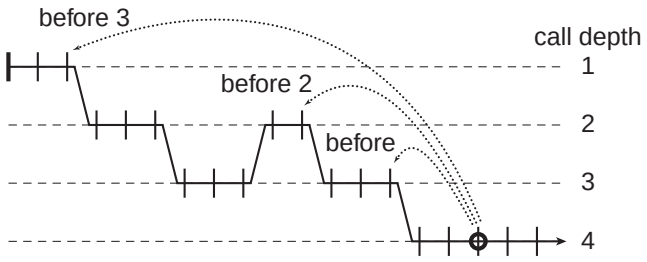


Figure 10: Destinations of *before* commands.

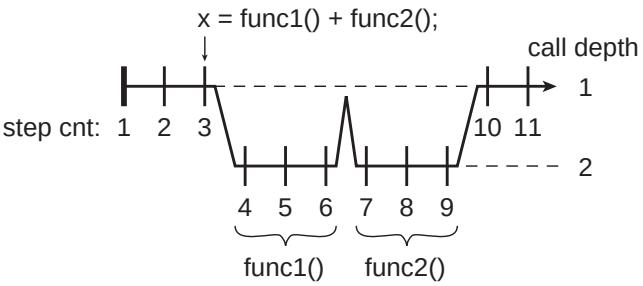


Figure 11: Subtlety when climbing out of a function.

ever the current call depth is equal to the desired depth of the *before* command. The last such value saved corresponds to the desired destination. The second pass now simply re-executes to the *step* count determined by the first pass.

This algorithm is capable of moving an arbitrary number of levels back up the call stack using only two passes. In practice we expect a user will often incrementally *before* back one level at a time, and we optimize for this case by identifying the subsequent *before* point during the second pass. This then eliminates the need of the first pass for each repeated *before* command.

2.6 Until & Buntil

A valuable feature of a debugger is the ability to run until a specified variable reaches a desired value. Unfortunately this has been unbearably slow in traditional debuggers (a slowdown factor of 85,000 has been reported for dbx[13]). Wahbe, Lucco, and Graham[13] demonstrated an efficient technique for implementing "data breakpoints" using code augmentation to monitor memory updates. They inserted code at all memory write instructions to check accessed addresses against a two-level segmented bitmap representing the collection of monitored addresses.

We have developed a simpler, although not as general implementation, by replacing our *step* counters with a test to monitor the value at a single memory location. We thus only test for changes in a data value at our stepping points. This is adequate for finding the exact statement at which a data value is changed, but it doen't identify the assembly language instruction.

We provide movement commands of the form "until x==10" to continue until a specified variable reaches a specified value, and commands of the form "until x" to continue until a specified variable changes.

Buntil is our analogous backwards movement. This may be the most valuable movement of this debugger. With a single command the user can find where a variable took on an observed erroneous value. Such errors can otherwise be extremely difficult to find if they result from an apparently unrelated operation such as a pointer problem or overrunning an array's bounds.

At every *step* point our counter routines monitor the value at the address of the specified variable and test to see if it matches the desired condition. To perform a forward *until* we stop at the first match. To perform a *buntil* we use the first pass to record the last *step* point at which the condition was satisfied (as we re-execute forward), and then a second pass to re-execute to that point.

Our implementation does have one important advantage over that of Wahbe, Lucco, and Graham's in that ours is able to stop at a specified value for a variable, whereas theirs is only able to detect the modification of a variable.

2.7 Undo

Undoing movements in bdb is easy using the underlying time line provided by the step counter. On the undo stack we save the initial step count before performing each movement command. To undo a movement, we simply retrieve the original step count from the undo stack and than either execute forward to that point, or re-execute to get back to it.

3 Implementation Subtleties

There are a number of subtleties in getting the movements precisely correct. Figure 11 shows one example. The statement "x = func1()+func2();" occurs at step 3 in the diagram. The execution of this statement is spread out over the region between step 3 and step 10: the first call occurs at step 3, the saving of its return value and the call to $func2$ occurs at the spike between the steps 6 and 7, and the final sum and assignment to x occurs just prior to step 10. The spike is drawn to indicate a possible phantom stopping point.

The question arises should a forward movement leaving $func1$ (or a backwards movement leaving $func2$) stop at the spike? For instance, should a *step* from point 6 stop at the spike, or stop at point 7? We decided to stop only at our counted stopping points between statements. For us, a *step* from 6 stops at 7, and a *next* or *finish* from 6 returns to the calling function and stops at 10.

To uphold this rule, our implementation of *previous* requires 3 re-execution passes for the special case of moving back past a spike, such as from point 7 to its destination at point 3. In the example, when our first re-execution pass hits the function exit following 6, it resets the "top" point to 7. Then when bdb notices that the "top" point is at the current position, it knows that a spike occurred. It then raises the base depth and re-executes another "first" pass, which properly identifies the desired destination at point 3.

Traditional debuggers differ on the matter of stopping at the spike. Gdb stops at the spike only for *finish*, whereas Microsoft Visual Studio always stops at the spike.

We feel that it is actually preferable for *finish* and *before* to stop at the spike because it makes it easier to get into the second function: one could *step* into the first function, *finish* that function, and then *step* into the second function. We have designed but not yet implemented counter routines to accomplish this.

The key idea is to have the function exit counter test for the stopping condition and trap back to bdb. We would then use the traditional debugger technique of inserting a

Forward Movements

command	slowdown
step	1.95
continue	1.95
next	2.48
finish	2.33
until	2.34

Backward Movements

command	1st use slowdown			repeated use
	1st pass	+	2nd pass	slowdown
bstep	1.95			1.95
bcontinue	1.95	+	1.95	1.95
previous	2.49	+	1.95	1.95
before	2.20	+	2.20	2.20
buntil ==	2.71	+	1.95	(always 2 passes)
buntil !=	3.15	+	1.95	(always 2 passes)

Table 1: Worst cases of measured execution slowdown factors for each movement.

temporary trap instructions at the return address and then continuing forward to that trap. This would not suffer from the performance problems encountered by traditional debuggers when finishing a recursive call because our counter is able to advance past all of the premature function returns.

The spike is not at an exact counted *step* point, and thus on our undo stack we would need to note the special circumstance by which we arrived at this point so that a future *undo* would return to the spike.

4 Overhead of Counter Routines

The overhead of our counter based movement algorithms varies based on the complexity of the code statements in the program being debugged. In particular, heavy use of library functions lends itself to lower overall overhead because more work is done between user level stepping points. We used 5 recent student assignments as test cases. These were: an anagram finder, a cryptographic deciphering program, a dynamic programming algorithm for finding the edit distances between proteins, an iterative successive over relaxation code, and an X-Windows fern drawing program. The most important performance measurement is the overhead of the basic *step* counter because this is used for normal forward execution. For the five student assignments, this overhead was 94%, 31%, 95%, 56%, and 0% respectively. (The overhead was negligible for the fern program because its execution time is dominated by X-Windows related system calls.) For the SPEC CINT95 benchmarks 129.compress, 130.li, and 147.vortex we measured overheads of 95%, 124%, and 103% respectively. We calculated these overheads relative to the performance when compiled for normal debugging. Our system starts with the output of "gcc -g", and thus our overhead is in addition to the normal debugging performance degradation.

Table 1 shows the execution slowdown factors for all of bdb's supported movements for the edit distance test case, which was the one that showed the greatest slowdowns among our students assignments. In actual use, for any short movement (which is the majority of movement commands) bdb responds instantaneously on a human timescale, and thus for short movements the slowdown factors are unimportant. We care about slowdowns only for long running movements. All of these results were gathered over the entire execution of the test programs. This was done, for example, with commands such as "step 1000000000", or by setting a breakpoint at the end of the program.

For forward movements, the slowdown is incurred over the distance moved forward. For backwards movements we have reported the slowdown factors for the first and second passes separately. These slowdown factors are incurred over the re-execution interval. For some movements the slowdown factors of the two passes differ because different counter routines are used. For example in the case of *previous*, the first pass uses the *previous* counter for the entire re-execution interval, but the second pass uses the simpler *step* counter for the majority of the interval until the last descending call chain on which it uses the *previous* counter.

It is difficult to compare our results to those of related projects because most of them provided at best only limited performance measurements, and admittedly most of these projects were more interested in providing backwards looking functionality than in the performance aspects. We thus compare ourselves to the most relevant numbers available to us. Netzer and Weaver[9] built an execution replay debugger for long running programs based on program tracing with an emphasis on minimizing the size of the trace file. They reported slowdown factors ranging from 1.75 to 7.0. Feldman and Brown[3] built a debugger based upon frequent (and fast) checkpointing and then program interpretation to move forward. Their interpreter incurred a slowdown factor of 140. Our slowdown factors compare favorably to both of these, which suggests that re-execution is a more efficient methodology.

From our experience using bdb, we feel the performance achieved by our counter based movements is more than adequate. However if one wished to push the performance level to its limits, a more aggressive implementation could reduce the overhead substantially. In an experiment we applied the following optimizations to the basic step counter: removing the indirection on counter calls (by instead copying the desired counter routine to the fixed call site), using a streamlined calling convention, using dedicated registers to hold counter values (as done by Mellor-Crummey & LeBlanc[7] and Wahbe, Lucco & Graham[13]), and counting down to zero (also as in[7]). Together these reduced the overhead by 75%. The overhead of the *step* counter in the edit distance application was reduced from 95% to 24%. This compares to an overhead of 10% for Mellor-Crummey's inline software counter and 42% for Wahbe's data breakpoints. We offer these numbers not to say that our implementation is slower or faster than these implementations, since clearly they are all doing different things, but rather to indicate what is achievable to a reader who is interested in maximum performance.

4.1 One Pass Algorithms

Another source of possible performance improvement might come from using one pass algorithms for some of the backwards movements. For instance by maintaining an array of breakpoints counts, one for each location, we could eliminate the extra pass for *bcontinue* that is needed after the user changes the set of breakpoints. By maintaining a stack of the last points at each call depth, we could eliminate the extra passes for *previous* and *before*.

Figure 13 shows the information needed by a 1 pass *pre-*

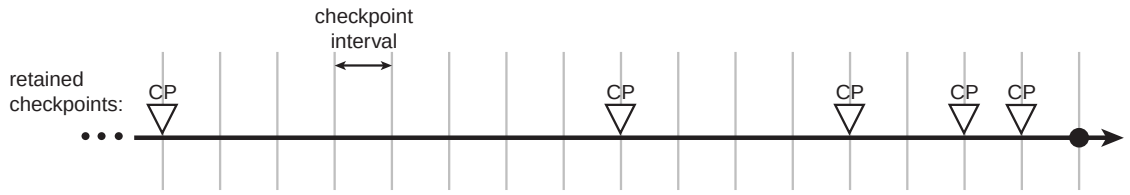


Figure 12: Checkpoints are thinned to leave an exponentially increasing series of intervals behind the current position.

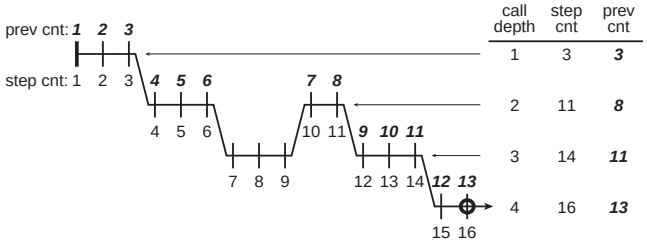


Figure 13: The 1 pass *previous* algorithm saves the last *step* and *previous* values at each call depth.

vious algorithm. At each call depth it stores the values of the *step* and *previous* counters corresponding to the last point at that depth. Upon exiting from a function our two pass algorithm discarded the *previous* count, recorded the "top" point, and started counting afresh. Instead the 1 pass algorithm can simply retrieve and continue counting from the correct *previous* count recorded before the intervening function call.

The drawback of these one pass algorithms, however, is that all of this additional information would need to be gathered during regular forward execution, just in case it was needed later. Fast forward execution is more important.

For long running programs, the most important performance measurement is the slowdown factor incurred by the basic step counter when executing forward. For finding problems far into a program's execution, a user is likely to either insert a breakpoint at a rare case or in an error detection routine, or to let the program run until it encounters a fault or spurious exit. At this point they would use short backwards movements to investigate what happened. The long forward execution just uses the *step* counter.

This is meant to be a practical debugging technique that can be used on real programs running for minutes, hours, or even days (such as a network server). Checkpointing (discussed next) will allow the user to quickly move back to recent events even after having executed for hours. For events further in the past, our checkpointing algorithm will provide response times proportional to the distance moved back.

5 Checkpointing

We expect execution time in a traditional debugger to be roughly equivalent to the temporal distance moved forward, and we expect short movements to be performed without noticeable delay. This is what we consider "efficient" performance; the only time a movement incurs noticeable delay is when the stopping point is so much farther along the execution path that the delay incurred would be expected, and these delays should be proportional within a small factor to the distance move forward. Traditional debuggers provide this most of the time, but fail to provide good performance for cases that cause them to execute a large number of traps back to the debugger as discussed in Section 1.2. Bdb's counter based methods perform all forward movements (including those case that are difficult for a traditional debugger) efficiently and accurately.

We likewise feel the same performance goal should be applied to backwards movements. Movements a short distance back should be performed without noticeable delay, and longer movements should take time roughly equivalent to the distance moved back. We have used checkpointing to address this goal.

We create checkpoints at regular intervals by forking the process being debugged. By using fork to create checkpoints, we take advantage of the operating system's efficient copy on write policy. For performance measurements we used a checkpoint interval of 1/10 of a second. In the for worst-case for the student applications this incurred an additional performance overhead of 7%, and on average an overhead of only 3%. For the much larger SPEC applications the overhead ranged from 9% to 45%. Increasing the checkpoint interval to 1 second brought the checkpointing overhead for the SPEC applications down to a range of 1% to 14%

Short backwards movements only need to re-execute in the last interval, and thus all short movements will finish quickly. For example, our slowest movement "buntil !=" on our worst case application presented in Table 1 has a two pass algorithm with a total slowdown factor of 5.1. Relative to its forward execution with a slowdown factor of 1.95, for a 1/10 of a second checkpoint interval this worst case for a short movement would require only 0.26 seconds.

5.1 Exponential Checkpoint Thinning

To avoid overwhelming the processor with checkpoints, we start thinning them out as they become older. Figure 12 shows a neatly thinned set of checkpoints representative of what our checkpoint thinning algorithm will aspire to. Our goal is to create a small set of checkpoints from which we can re-execute to any earlier point in a time proportional to the distance moved back. In the figure the checkpoints are retained at: 1, 2, 4, 8, and 16 intervals back from the current position (represented by the black dot). These intervals would continue to grow exponentially as powers of 2 as they went further back in time until the beginning of the program's execution. Observe that the size of each checkpoint interval in this example is equal to the distance between the end of that interval and the current position. Hence any prior execution point is contained within a checkpoint interval that is no longer than distance between that exectution point and the the current position.

Our thinning policy, is designed to achieve a set of checkpoint intervals that maintains this property that no execution point ever be in a checkpoint interval whose size is greater than the distance from the current position back to that point. These checkpoint intervals are adjusted as

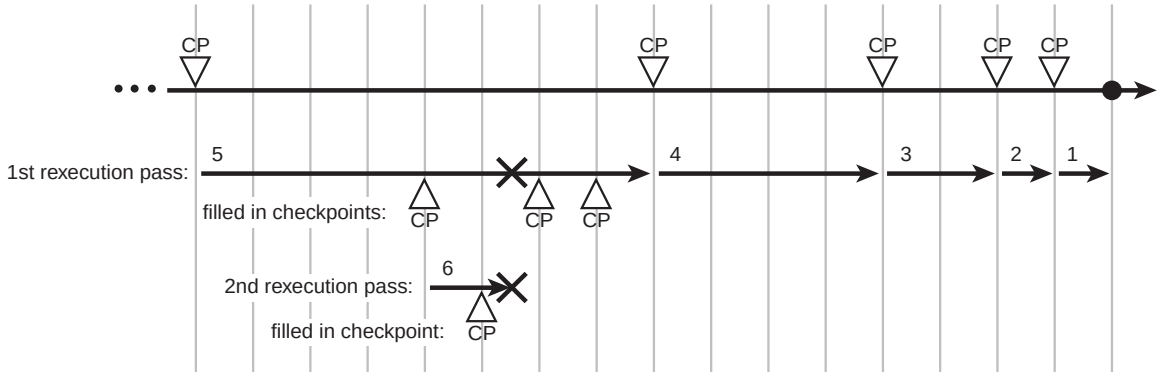


Figure 14: A backwards movement explores the checkpoint intervals in reverse, starting from the current position, until it locates the closest interval containing the event of interest. For movements requiring two passes, the second pass is only performed on the interval containing the stopping event.

we dynamically execute forward. Each time we reach the start of a new checkpoint interval, we scan the current list of checkpoints and thin out any checkpoint such that the interval between checkpoints would not grow larger than the distance from the end of the resulting combined interval to the current execution point. This does indeed create a series of exponentially larger checkpoint intervals as they get further back, although since these are created dynamically on the fly as we execute forward, they are rarely as neatly proportioned at exact powers of 2 as in the example.

5.2 One Pass Movements

Backwards movements explore these checkpoint intervals in reverse to find the closest preceding interval containing the point of interest. By our interval construction, the interval containing the stopping point cannot be larger than the distance moved back, and also the sum of the sizes of the other intervals explored, as we work our way back to the desired interval, is also less than or equal to the distance moved back. Together this bounds the sum of the sizes of the checkpoint intervals explored to be less than a factor of 2 times the temporal distance moved back.

5.3 Two Pass Movements

The preceding argument applies to single pass backwards movements. For two pass backwards movements we must consider the cost of the second pass. This will be explained with the help of Figure 14. This figure shows the set of exponentially thinned checkpoint intervals from the previous figure along with numbered arrows indicating the order of re-executions performed when seeking the point marked with the X. This point could be any point sought by a two pass movement: a location breakpoint, a *previous* point, a *before* point, or a data value breakpoint. We will refer to it as a breakpoint in our discussion.

The first re-execution pass is seeking to find the closest preceding interval containing the breakpoint. It explores the checkpoint intervals in reverse order until an interval containing a breakpoint is found. (If we were looking for a counted breakpoint, any intervals found containing fewer than the desired count would simply deduct the number of breakpoints found in that interval from the desired total.) Only one breakpoint is shown, but the interval could in fact contain multiple breakpoint occurrences. If so, we want the last one (or the desired number back in case of a counted movement). In any case, since we do not know how many will occur within the interval, the first pass will count the number of occurrences, and the second pass will execute to the desired count.

As we re-execute each interval, we fill in any missing checkpoints for that interval. When we complete the re-execution of a large checkpoint interval, we will thus have a fresh set of checkpoint intervals that obey our property of none being larger than the distance back from the current point (which is now at the end of the interval just filled in). This is shown in the figure only for checkpoint interval number 5. Checkpoints are in fact not needed for later execution points because any later point can be reached through forward execution in linear time, and thus by the time the debugger finishes interval 5, it may have already discarded any filled in checkpoints for intervals 1 through 4 as well as even the checkpoints separating those intervals.

Given this filling in of checkpoints, for the second pass of a two pass movement algorithm we only need to re-execute in the single small checkpoint interval found to contain the breakpoint. In the example this is interval 6.

In the case shown, the total re-execution is 17.5 intervals, and the distance moved back is 10.5 intervals, for a net re-execution factor of 1.7 times the distance moved back. This still falls within our factor of 2 bound. The only place where it can exceed this bound is within the last (in the forward direction) sub-interval of a filled in interval. For example if the destination point was just a little bit prior to interval 4, the temporal distance moved back would be $8 + \epsilon$ and the total needed re-execution would be $17 - \epsilon$. This leads to a bound for 2 pass algorithms of a factor two times the temporal distance moved back plus one minimum checkpoint interval.

5.4 Faster and Slower Cases

The above discussion is valid for the first backwards movement executed. Unfortunately with multiple backwards movements our system can experience performance anomalies that do not meet our goals. One such case occurs after the user has stepped back a long distance to a point near the start of a large checkpoint interval (such as the start of interval 4). That movement actually performs very well, requiring re-execution of only 1 times the distance moved back, rather than the upper bound of 2 times. However a subsequent short step back into the preceding large check-

point interval (interval 5 in this case) requires re-execution of that entire interval, thus paying the price for our good luck on the preceding movement. The performance on the short movement considered by itself does not meet our performance goal.

There are also cases where we perform much faster than expected. The debugger maintains any potentially useful counter information gathered for each checkpoint interval. For example, for breakpoints the debugger records the number of breakpoint visits and a version number representing the set of breakpoints at the time the counts were gathered. Using the example from Figure 14, if breakpoint counts had been gathered during forward execution and we were now looking for the same set of breakpoints in reverse, the debugger would know that no breakpoints occurred in intervals 1 through 4, and it would also know the number of breakpoints occurring in interval 5. It would them only need to perform the second pass re-execution on interval 5. The total time for this being about half the temporal distance moved back.

These saved counts are combined and split as checkpoints are removed or inserted into checkpoint intervals. They can also sometimes be used for faster than expected forward movements, if for example the debugger already knows that no breakpoints occur in a certain interval.

5.5 Fast Undo

A final feature we provide is a fast undo. When starting a new movement, we always retain the closest checkpoint preceding the starting point. If asked to undo the movement, we can then execute to the starting point quickly from the retained checkpoint. Thus if the user accidentally continues from some point, seeking some event that does not occur, when they realize it has taken too long, they can interrupt the debugger, issue an undo command, and immediately be back where they were.

5.6 Space Usage

The set of checkpoints grows logarithmically as a function of the execution time, and the checkpoints (created by forking a process) are each the size of the process. We expect in practice that there will be much economy in space usage gained by an operating system that uses a copy on write policy, for then the space needs will only involve the page tables and those pages that have changed between checkpoints. We have not evaluated the extent of this benefit.

There are tradeoffs that could be made to decrease the number of checkpoints needed. For example we could increase the checkpoint interval to 1 second. This would lead to a noticeable but tolerable increase in the re-execution time for short backwards movements (in the range of from 1 to 3 seconds). This would also greatly reduce the checkpoint creation overhead. For situations involving trying to locate sporadic bugs that appear after days of execution, a checkpoint interval of 1 minute would seem reasonable. Once the debugger stopped on an error, we could then reduce the checkpoint interval and in 1 minutes time fill in the checkpoints for the last minute long interval by re-executing that interval.

We also might set a limit on how far back a user is able to move, perhaps a few minutes, and then discard any checkpoints older than that limit. This would give us a constant bound on the number checkpoints. Alternatively, we could set a constant bound on the number of checkpoints and then at points when the need for checkpoints exceeded the bound, we could relax the factor of 2 on checkpoint thinning. This would gradually increase the factor of 2 bound on re-execution time relative to the temporal distance moved back.

There certainly are other reasonable options for managing the set of checkpoints. Tolmach and Appel[12] in a project with many similarities to our own, chose to manage their checkpoints as a cache. This was motivated by their argument that user activity is likely to be clustered around certain points along the execution time line, and that caching would retain those checkpoints that were used.

6 I/O Replay

An important requirement for successful re-execution is that the program re-execute deterministically. Cases where we might get nondeterministic behavior include seeding a random number generator based upon the time of day, or modifying a file so that we read different values from it upon re-execution.

We provide the ability to capture the return values from system calls and replay them during re-execution. For example, on re-execution of a system call to get the time of day, we return the time from the original system call, and thus the program would seed the random number generator the same. On re-execution of a read, we return the original data read, even though it may not actually be in the file anymore. For a write we actually only need to record and replay the returned status value.

UNIX provides a mechanism for trapping to the debugger upon the entry and exit of system calls by the child. Unfortunately this proved inadequate for our purposes and we had to resort to a more cumbersome method using explicit trap instructions before and after the system call instruction.

None of our test cases used for performance evaluation actually behave nondeterministically, and by default I/O logging is disabled. We report here the performance impact when I/O logging is enabled.

For the first 4 applications, which have varying but not overwhelming I/O usage, we measured their I/O logging overheads as 3.5%, 72%, 75%, and 0.1%. However the fern program, which is completely I/O bound, had an overhead of 5100%, which translates to a slowdown by a factor of 52. Replaying from the I/O log incurs roughly half the overhead incurred when creating it.

There are also situations in which our re-execution performance may actually be much faster then the original program's execution time. In a network server application, or an interactive application, a large fraction of the wall clock time may be expended waiting for events such as messages or key clicks. During re-execution we replay the events, but we don't replay the waiting time. For a simple text editor, we could probably replay hours of user time in a few minutes or even seconds.

A large portion of programs written today are interactive graphical applications. These are both nondeterministic (because they are interactive) and potentially I/O bound (because of the graphics). Further improvements in our I/O logging are clearly needed to accommodate these applications,

Furthermore, our I/O logging and replay occurs at the process boundary. In an X-Windows application, when we back up the state of the process, we can see what happened within the process, what the values of its variables were, and what X-Windows calls were made. But we don't see the graphics window redrawn. This is outside the process

being debugged, and thus it is only affected by the I/O performed by the original process and not by the re-execution processes. Backing up the graphic state along with the internal process state would be a valuable addition. This was nicely done in a reversible LISP debugger by Lieberman and Fry[6].

6.1 Difficulties

I/O logging is a complex task. There are 262 Unix system calls. Some are simple such as "getpid", and just have register return values. Others such a "read" require the buffering of possibly large inputs. The "readv" system call has even more complex buffering requirements due to its vector of buffers, and other system calls such as "ioctl" have been used for a myriad of different purposes and are a quagmire of special cases.

For most system calls we capture their return values when the original program executes the system call, and we replay those captured values when re-executing the system call. However there are some system calls that actually need to be re-executed. For example, "obreak" is used to change the top of the heap when more space is needed for dynamic memory allocation. We must re-execute it so that the re-execution process will have its heap extended as well.

Of the 262 system calls, we have implemented (or partially implemented in the case of "ioctl") I/O replay for only 35 of them. These are the ones that we have seen used in our test programs. Much further work could be done on the I/O replay aspects of bdb.

7 Related Research

There is a diverse body of research addressing many of the issues brought together in this project. Mellor-Crummey and LeBlanc[7] developed a fast software counter for program replay. Kessler[5] used code augmentation to provide fast breakpoints for profiling. Wahbe, Lucco and Graham[13] investigated fast data breakpoints during forward execution. Feldman and Brown[3] based their reversible debugger on checkpointing. Pan and Linton[10] discussed logging system calls and replaying them for deterministic re-execution.

The most common approach to producing bidirectional debuggers has been to create a history log of all changes to variables[1, 2, 6, 8, 9, 11, 14]. Two of the best such projects were those by Lieberman and Fry[6] and by Moher[8]. Both provided a rich set of bidirectional movements, but were limited to short running programs because of the rapid growth of the history log. They also were based on interpreters so they incurred large execution slowdowns. Agrawal[1] and Netzer[9] both addressed the history log size problem by condensing the changes to a coarse set of points, but at the cost of less flexibility in backwards movements.

Re-execution has been used primarily for deterministic forward replay of parallel programs[4, 10]. Feldman and Brown[3] built a debugger for sequential programs based upon frequent (and fast) checkpointing, in conjunction with an interpreter to move forward from checkpoints, but they were hampered by the slow speed of interpretation. The most significant use of re-execution to date for bidirectional debugging was an ML debugger by Tolmach and Appel[12]. They used a step counter to provide *step* and *bstep* movements with a reported average slowdown of 2.7. *Continue* 1, *bcontinue* 1, and *next* 1 were provided by using a large bit vector with one bit for every source location and a process of using a logarithmic number of re-execution passes to binary search over time intervals. This binary search was performed not only for backward breakpoints, but also for forward breakpoints! Their bit flag and binary search algorithm also limited them to moving by a single breakpoint at a time, so efficient counted breakpoints were not possible under their system. They gave no performance measurements for these more complex movements.

8 Conclusions

We have developed a combination of techniques and demonstrated through a working implementation that it is possible to build a bidirectional program debugger, that is practical, efficient, precise, and applicable to a very broad spectrum of programs.

Central to this accomplishment is our technique of embedding calls to counter routines at stepping points and at function entry and exit points throughout the program being debugged. As the user issues debugging movement commands, we can efficiently switch the entire set of calls to a group of customized counters designed to implement the desired movement. We have developed simple counter routines to provide a complete set of both forward and backwards movements: *step n*, *continue n*, *next n*, *finish n*, *until* ==, *until* !=, *bstep n*, *bcontinue n*, *previous n*, *before n*, *buntil* ==, *buntil* !=, and a general *undo*.

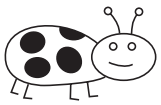
The efficiency of our debugger arises from our use of program re-execution to reconstruct earlier program states along with periodic checkpointing to limit the amount of re-execution needed. We have developed a method of exponential checkpoint thinning that allows us to locate and move back to any nearby points quickly and to reach points further back within a maximum time bound of roughly twice the temporal distance moved back. Our exponential thinning of checkpoints limits the number of checkpoints needed to a logarithmic function of the range of backwards movements.

Finally I/O logging and replay allow us to ensure deterministic re-execution for those programs that otherwise might re-execute in a nondeterministic fashion.

A bidirectional debugger such as bdb can alleviate much of the tedium in using a traditional debugger by greatly simplifying and expediting the process of tracing the cause of a bug backwards from the point where it becomes manifest. Furthermore, the trepidation of stepping past something of importance, when using a forward only debugger, is lessened since undoing incorrect movements is now simple and immediate. We hope that the success of this project will spur widespread incorporation of bidirectional capabilities into commercial debuggers.

9 Acknowledgments

We gratefully acknowledge the hard work of the graduate students who helped build this project: Robert Zulawnik, Steve Dorato, Richard Best, and Charles Carr. We appreciate the enthusiasm and encouragement of early viewers of this research, and we thank NSF for their support through grant CCR-9619456. Finally, we thank the PLDI reviewers for their thoughtful comments and suggestions.



References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An Execution-Bracktracking Approach to Debugging. *IEEE Software*, 8(3):21–26, May 1991.

[2] R. M. Balzer. Exdams: Extensible debugging and monitoring system. In *Proc. Spring Joint Computer Conf.*, pages 567–589. AFIPS Press, Reston, VA, 1969.

[3] Stuart I. Feldman and Channing Brown. Igor: a system for program debugging via reversible execution. In *Proc. SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 112–123, Jan. 1989.

[4] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. In *Proc. SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 163–173, Jan. 1989.

[5] Peter B. Kessler. Fast breakpoints: Design and implementation. In *Proc. SIGPLAN'90 PLDI Conf.*, pages 78–84, June 1990.

[6] Henry Lieberman and Christopher Fry. *Software Visualization*, chapter ZStep95: A Reversible, Animated Source Code Stepper, pages 277–292. MIT Press, 1998.

[7] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *ASPLOS-III Proceedings*, pages 78–86, April 1989. Appeared as SIGPLAN Notices 24(Special Issue).

[8] T. G. Moher. PROVIDE: A Process Visualization and Debugging Environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[9] Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proc. SIGPLAN '94 PLDI Conf.*, pages 313–325, June 1994.

[10] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *Proc. SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 124–129, Jan. 1989.

[11] Daniel G. Shapiro. Sniffer: a System that Understands Bugs. Master's thesis, MIT Dept. of EE&CS, 1981.

[12] Andrew Tolmach and Andrew Appel. A debugger for standard ML. *J. Functional Prog.*, Jan. 1993.

[13] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proc. of the SIGPLAN'93 PLDI Conf.*, pages 1–12, 1993.

[14] M.V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, September 1973.