Bisection Debugging

Thomas Gross

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Institut für Computer Systeme
ETH Zürich
CH 8092 Zürich

Preliminary version

Abstract

This paper introduces the *bisection debugging* model. The key idea is to use a debugger to identify the semantic differences between *two versions* of the same program. The debugger leads the user (programmer) to the region of the source code that is different between the versions and effects erroneous behavior. To implement bisection debugging, a debugger must insert checkpoints around regions of the program that are determined to be different (by, e.g., a file differencing program). To compare the state of both computations, it is necessary that there are no constraints on where a breakpoint can be set, and the debugger must be able to solve all data value problems at this breakpoint. The quality of the file differencing program has an influence on the execution speed of the debugged program, but not on the correctness of the method. This debugging model is useful if there exists already a correct version of a program (in the sense that its output is acceptable) and subsequent changes have modified this program so that the output is no longer correct.

1 Introduction

A common situation in program development and maintenance is that a change to the program introduces a bug. That is, an earlier version P_{old} of the program under development is deemed to be correct, and a later version P_{new} is considered to be incorrect. There are many reasons why it is needed to modify a correct program – to add functionality, to port it to a different version of the compiler or runtime system, to remove bugs for some input sets, etc. When such changes are made, software developers are concerned about not breaking the parts of the program that work so far, and regression testing (or other testing approaches) are commonly used to guard against such mistakes. However, if a bug is inserted, then the programmer wants to identify (and remove it) as soon as possible.

This debugging situation is somewhat different from other scenarios during development when *no* reference version exists. Here, we assume that there exists a version P_{old} that produces a correct result for some input I so that $P_{old}(I)$ is acceptable and $P_{new}(I)$ is not acceptable. Note that $P_{old}(I')$ for some other input I' may be incorrect, and that $P_{new}(I')$ is correct. All we care about is that a bug has been inserted into P_{new}.

Since P_{new} and P_{old} are versions of the same program, it is possible to identify their differences using either a separate utility tool or obtaining them directly from the program development environment. A user is then interested in finding the earliest point in the execution of the two versions where the data space differs – once this point has been identified, then it is possible to unravel the situation.

Skilled programmers have used this technique for a long time by manually comparing the states of two executions of different versions of this program. However, inserting by hand breakpoints in those places

where the debugger user suspects that there exists the potential for differences is cumbersome if the program is large. A recent paper [1] described the implementation of such a system to identify problems occurring in the porting of applications.

2 Example

Consider the simple loop shown in this example:

```
for i:= 0 to n do
  begin
  a[i] = i;
  end;
```

If this code is modified to introduce a guard for the assignment statement in the loop

```
for i:= 0 to n do
  begin
  if (b[i] <> 0) then
     a[i] = i;
  end;
```

(and this region of code is executed prior to code related to other changes, then we want to stop execution if there exists an i' such that $\mathtt{a}_{new}[\mathtt{i}] = \mathtt{a}_{old}[\mathtt{i}]$ for all $i < i'$ and $\mathtt{a}_{new}[\mathtt{i}'] <> \mathtt{a}_{old}[\mathtt{i}']$. A simple bisection debugger may insert a breakpoint before and after the loop in each version of the program and compare the value of `a` in both executions. A debugger that has a better understanding of the program structure would insert a test inside the loop, as indicated below for P_{old}

```
for i:= 0 to n do
  begin
  // test //
  a[i] = i;
  end;
```

and for P_{new}

```
for i:= 0 to n do
  begin
  if (b[i] <> 0) then
     // test //
     a[i] = i;
  else
    // test //
  end;
```

3 Problem

Given two versions P_{new} and P_{old} of a program with different behavior, finding the textual differences in the source files is often not enough to identify the cause of the behavior change. In addition to edits, there may be additional files for P_{new} in the source code depository, or files have have been merged or deleted altogether. We assume that at least the highest-level function (`main` in C/C++) has retained its name. (C

and C++ require a specific name, Java imposes some constraints on the class name that in practice guarantee that this requirement is met.)

Also, to concentrate on the novel aspect of automatically comparing the execution states of two versions of a program, we require that all programs are well-structured (i.e. do not contain arbitrary control transfers). Most C/C++ programs and all Java programs satisfy the constraints.

Figure 1 illustrates the situation; a shaded region indicates that these regions of the program have changed in P_{new} relative to P_{old} or that have been removed from P_{old}. However, not all changes may have an adverse effect with regard to our input set (I).

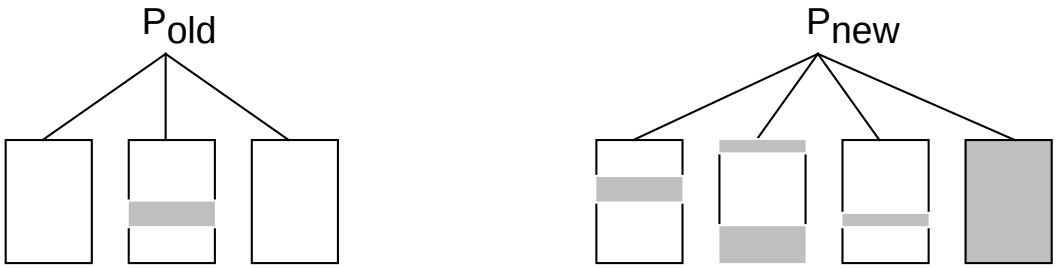

Figure 1: Sources for P_{old} and P_{new}.

To identify the changes that matter, we want to divide the source files in half and see if the changes to the first half result in a different state of program execution for P_{new} (compared to executing P_{old}). It is not critical that we divide the source exactly in half: One one hand, even if we divide the sources so that both regions have an equal number of (non-comment) characters, there is no guarantee that the execution time mirrors this division. Also, on the other hand, it may not even be possible to achieve an even division with regard to space or time.

Ideally, we are able to divide the sources so that the execution of each region forms a strongly connected component, with a single control transfer from region 1 to region 2. Figure 2 depicts this idealized situation; a control transfer is a transition into or out of the hatched region in the execution diagram.

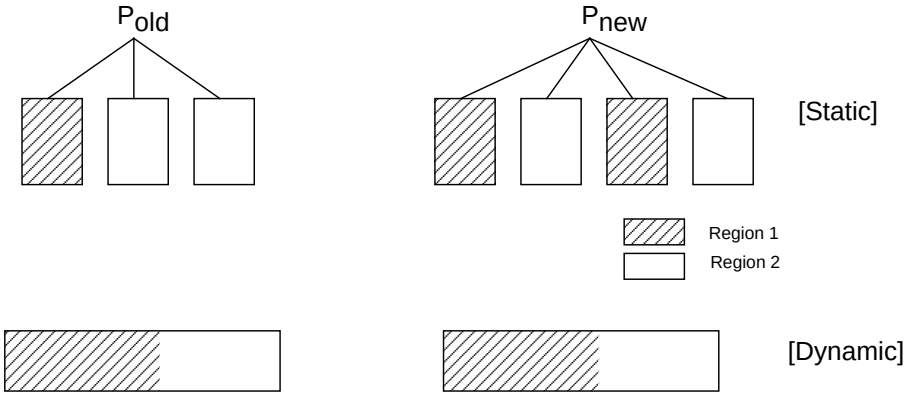

Figure 2: Idealized partitioning of sources for P_{old} and P_{new}.

However, in practice it is seldom possible to obtain a clean separation as depicted by Figure 2. Instead, a bisection debugger must deal with the situation that there are multiple control transfers between the two regions, as depicted by Figure 3.

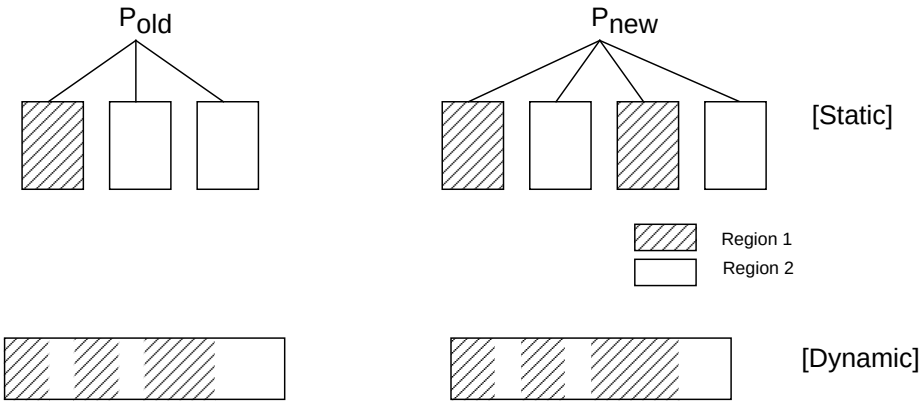

Figure 3: More realistic partitioning of sources for P_{old} and P_{new}.

To capture (and compare) the changes to the state of the execution for P_{new} and P_{old}, it is imperative that debugger is able to stop at any point, as determined by the source code partitioner. At these breakpoints, the debugger must be able to inspect all source variables and must be able to retrieve their current values. (A value is current if the object value corresponds to the source code value [6].) Finding a solution to this problem is similar to the task faced by a debugger for optimized code. In both cases, the debugger must understand the flow of data.

If the computations in the first region did not introduce divergence between the states of the computation, then we assume that the second region is responsible, so we continue with this region. In either case, we now proceed as before: the selected region is again divided into two regions, and the search continues. Once we have narrowed the selection to either a function, basic block, or individual line of code, as selected by the user, the process terminates.

Notice that bisection debugging assumes that only the program makes (relevant) changes to the program state. This assumption is violated by programs that, e.g., interact with the operating system and have for the same input set different execution histories, depending on some interaction with the operating system. `volatile` data structures do not rule out the use of bisection debugger, however the technique described here is unable to deal with bugs that are introduced by the interaction with an outside agent (the operating system or another thread that shares part of the address space).

4 Identifying the checkpoints

We now turn our attention to identifying the places where to insert checkpoints. A checkpoint is a pair of breakpoints in the sources of P_{new} and P_{old} that requires a comparison of the execution states. If all source variables have the same values, then the computations have not yet reached the point of divergence. In this case, the debugger continues to the next checkpoint.

Figure 4 depicts the situation that the region of interest in P_{new} and P_{old} consists of exactly one basic block. Shading indicates that these two blocks differ. The debugger now has to insert a checkpoint at each transition point, i.e. those places where the computations in P_{new} and P_{old} differ. Notice that there may be no differences in the basic blocks; in this case only the checkpoints at the start and end of the block are required.

This idea can be extended to a situation where the regions of P_{new} and P_{old} that the debugger wants to compare have a different number of basic blocks. Figure 5 depicts a single basic block in P_{old} that is replaced by three basic blocks in P_{new}. (Notice that one of the blocks in P_{new} may be empty, or that one of

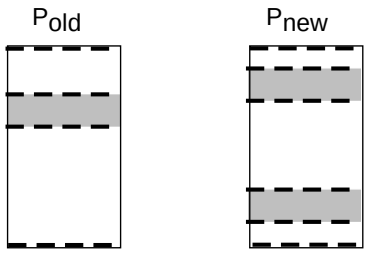

Figure 4: Differences in a single basic block.

them may be a loop or may represent a number of basic blocks.)

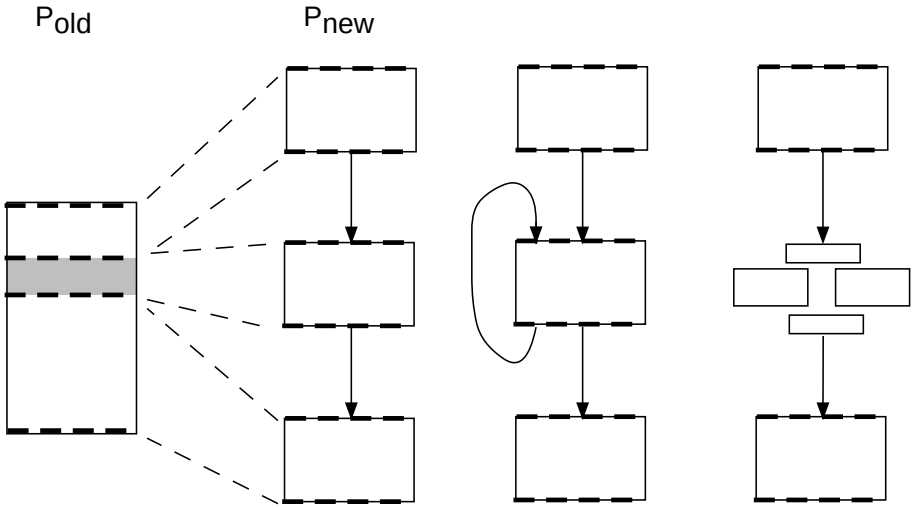

Figure 5: Expansion of basic blocks.

Again, in this case it is sufficient to introduce a checkpoint at any transition point in the program. Of course, the opposite case (three basic blocks in P_{old}, a single basic block in P_{new}) is also possible. The description of how to deal with this situation has been omitted from this summary.

Finally, we must deal with the situation that the regions in P_{new} and P_{old} contain several basic blocks. In this case, we can select an arbitrary partitioning, although one that attempts to match the textual differences will require fewer steps before the program is processed.

5 Related work

A prerequisite for this debugging model is the ability to debug globally optimized programs (for a realistic programming language). Only if the debugger can query and supply the state of the computation at an arbitrary place, then it is possible to use the bisection debugging technique. A number of papers on debugging optimized code have recently been published that demonstrate that it is now possible to debug fully (or at least highly) optimized code, even for languages as difficult as C [2, 3, 9]. However, the technology developed in these research projects has not yet found its way into commercial systems.

A recent paper [1] described the implementation of a relative debugger that allows a user to insert

breakpoints to compare the executions of two versions of a program on different target machines. The paper does not discuss the implications of (or constraints imposed by) code optimizers. The paper, however, points out a number of important observations for a multi-hosted environment. If the two versions of the program run on different target machines with possibly different implementations of floating point arithmetic computations, then the results of these computations may be different yet not point to an error. For example, if the order of summation is different on these machines, then rounding errors may produce different results. For this reason, we do not consider different target machines in this context.

Program slicing [4, 7, 5, 8] attempts to solve the inverse problem: given a breakpoint (and possibly a variable), find all parts of the program that were executed on the path(s) leading to this breakpoint. If the programmer has selected a specific variable, the slicing debugger may also highlight those parts of the program that may have set or modified the specific variable. The slicing approach can be useful (in the context of reconciling two versions of a program) if a the user has a good idea how the two versions differ. However, if many parts of a program are executed on those paths to the breakpoint, then the user needs assistance to compare the different paths in the two versions. This task, however, is much harder than comparing two different versions, because now we are given two slices (of two different but related programs).

6 Concluding remarks

This paper sketched an approach to debugging that assists a programmer in reconciling two versions P_{new} and P_{old} of a program. A bisection debugger partitions the sources of these two versions and attempts to identify the earliest point where the two computations diverge. This information then has to be used by the programmer to identify the real cause of the erroneous behavior.

Bisection debugging requires the ability to insert arbitrary breakpoints and to take a complete snapshot of the program state each each breakpoint. Since most modern compilers include optimizations, the above requirement implies that we are able to handle the problem of debugging optimized code; the debugger must be able to solve the data value problems at all breakpoints for the variables of interest. Since the changes in the versions may be local to a single basic block, the debugger must be able to handle both local and global optimizations. An approach that inserts breakpoints only at basic block boundaries is not sufficient.

One restriction is that the execution of the program is only controlled by the variables visible in the program (so that it is possible to executed P_{new} and P_{old} with the same input set I. As stated earlier, if the program contains references to `volatile` variables (or their equivalent), then this requirement is not met. A related comment concerns changes to the data space, i.e., the addition or removal of variables to the program. Since a bisection debugger attempts to point out where the computation performed by P_{old} differs from the computation performed by P_{new}, it is necessary that both P_{new} and P_{old} have (data) variables in common. If variables are renamed, then the debugger cannot correlate different definitions. If a variable has been deleted from P_{old}, then bisection debugging will identify the first place where a common variable is assigned a different value. Of course, it is possible that the paths through the program prior to this point are different (but they would only assign values to variables introduced newly into P_{new}).

Significant work remains to be done before we are in a position to experiment with a practical implementation of a bisection debugger. For an implementation, a good starting point would be the programming environment if this tool maintains a history of a program. As the program P is modified, the programming environment (or its editor) can record the deltas to earlier versions and thereby assist the debugger in identifying the regions of interest.

Acknowledgments

Woody Lichtenstein, then of Thinking Machines, now SGI, suggested the use of this technique for the debugging of parallel programs.

References

[1] D. Abramson, I. Foster, J. Michalakes, and R. Sosic. Relative debugging: A new methodology for debugging scientific applications. *Comm. ACM*, 39(11):69–77, Nov 1996.

[2] A. Adl-Tabatabai. *Source-Level Debugging of Globally Optimized Code*. PhD thesis, Carnegie Mellon University, 1996. CMU-CS-96-133.

[3] A. Adl-Tabatabai and T. Gross. Source-level debugging of scalar optimized code. In *Proc. ACM SIGPLAN'96 Conf. on Prog. Language Design and Implementation*, pages 33–43. ACM, May 1996.

[4] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proc. ACM SIGPLAN'90 Conf. on Prog. Language Design and Implementation*, pages 246–256, White Plains, June 1990. ACM.

[5] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Eng.*, 17(8):751–761, Aug 1991.

[6] J. L. Hennessy. Symbolic debugging of optimized code. *ACM Trans. on Prog. Lang. Syst.*, 4(3):323–344, July 1982.

[7] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proc. Conf. on Software Maintenance'93*, pages 386–395, Montreal, Sept 1993. IEEE.

[8] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Systems and Software*, 13(3):187–195, Nov 1990.

[9] R. Wismueller. Debugging of globally optimized programs using data flow analysis. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 278–289. ACM, June 1994.