

RELATIVE DEBUGGING USING MULTIPLE PROGRAM VERSIONS

David Abramson

and

Rok Sosič

School of Computing and Information Technology, Griffith University, Kessels Rd

Nathan, Qld 4111, Australia

E-mail: davida@cit.gu.edu.au

ABSTRACT

Incremental software development is the process of building new programs by modifying old ones. This method of programming is extremely common, and takes advantage of existing pieces of software which are known to be correct. The result is that there are many software versions in existence at the same time. To date, most software development tools have addressed only one aspect of incremental software development, namely, source code control. An equally important issue is in verifying that future versions of a program have the same dynamic behaviour as the original. Currently, there are no tools which address this issue. In this paper we introduce a new concept called relative debugging, which supports the incremental development of computer programs. Relative debugging makes it possible to compare the execution of two programs with the same functionality. The programs can run on the same computer, or on different computers connected by a network. They can differ in their source programming language or in implementation details. One of the programs usually serves as a reference version which produces the desired results. The execution of the other process is compared to the reference version in order to find discrepancies. We describe basic primitives for relative debugging and their implementation in a debugger called Guard. Guard is built on top of our portable debugging interface which provides a debugging platform for heterogeneous environments. The utility of Guard is illustrated by a practical example, in which a program written in Fortran is used to find errors in a version of the same code written in the C programming language.

1. Introduction

Program debuggers are tools which allow a user to control the execution of an application program through *breakpoints*, and to *examine* and *modify* the program state^{10,12,14,18,19}. Some recent significant extensions to conventional debuggers have been the addition of graphical user interfaces to improve the ease of use; data visualisation facilities to aid the interpretation of large and complex data structures; the concept of process groups to aid the management of many independent threads in parallel machines; and the addition of support for parallel and distributed debugging^{2,3,5,7,8,9,10,11,15}. In general, debuggers provide commands for the detailed examination of the internal state of a program. Most existing debuggers are geared toward the development of new programs. By dealing with each program in isolation, they offer only limited support for the maintenance and evolution of computer programs or for

the conversion of software from one machine or language to another.

However, when a program is being maintained or converted for execution on a new platform, a working version of the code already exists. By acting as a reference point, the working version can assist in locating the section of code in the converted program which produces faulty values. Currently, the version under development is compared to the reference version using ad-hoc techniques. Because the existing debuggers do not provide any supporting facilities, the most common method for comparing the execution of two programs is to run both programs under two separate copies of the debugger. The user must manually set the breakpoints, run the programs and visually compare the resulting program states on the screen. As a results, this approach is extremely error prone and tedious. A more advanced approach is to insert a number of output statements into both programs, run them on their respective computer systems, and then to compare the output using a file comparison program ⁴. This approach requires sufficient disk storage for the program output, which can be a problem for long running programs. Further, both programs must be modified and rerun. An additional limitation is that it is hard to extend this approach to take into account the semantics of data. For example, if floating point numbers are being compared, then simple file comparison programs may not be sufficiently flexible to detect when two numbers are equal.

In this paper we propose a new debugging paradigm, called *relative debugging*, which allows a user to compare the execution of a program with its reference version. Relative debugging does not require any modifications to the user programs and performs comparisons on the fly without requiring any disk storage. The comparisons take into account differences in data representations between programs, executing on different machines or written in different languages.

The paper begins with a description of relative debugging and issues in implementing relative debuggers. We then describe an implementation of a relative debugger, called Guard. Guard uses a portable debugging interface to provide relative debugging in heterogeneous, multilanguage environments. The key features of Guard are demonstrated in debugging a numeric program. The reference version of the numeric program, which is written in Fortran-77, is used to debug the same program, written in the C programming language. We conclude with some future directions for relative debugging and its application to parallel computer systems.

2. Relative Debugging

This section introduces relative debugging. It describes the basic functionality required of a relative debugger and discusses some implementational issues.

To verify the correctness of the program which is being developed, relative debugging exploits the existence of a reference program which is known to work correctly. At the beginning of the debugging session, the user formulates a set of assertions

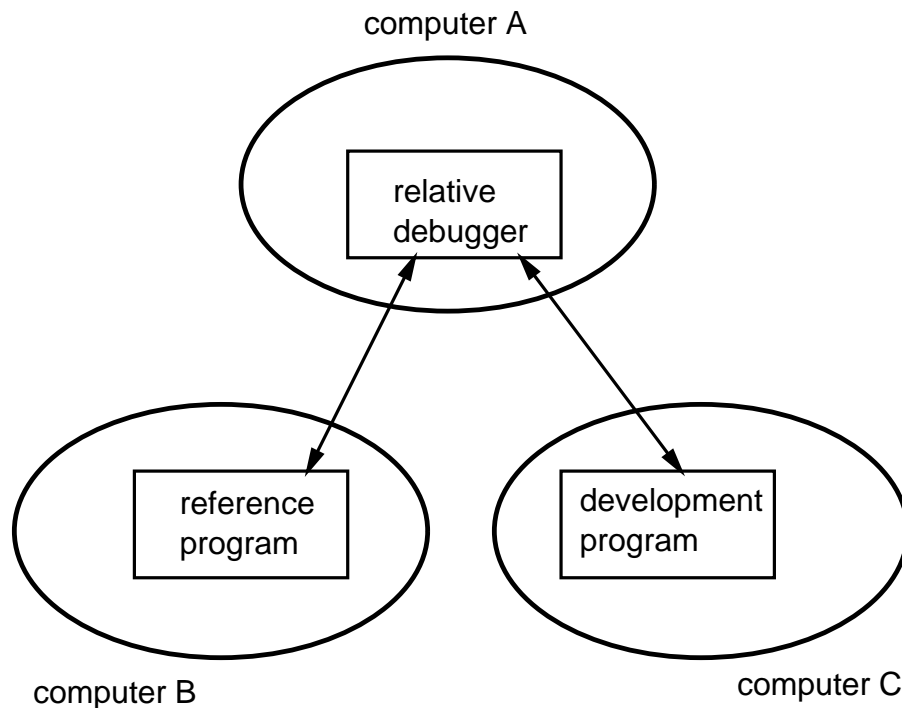


Figure 1: Relative Debugging

about key data structures in the reference and the development versions. The assertions state the locations at which these data structures should have the same values. Any violations of the assertions indicate an error. The relative debugger manages the execution of the development and the reference versions. It validates the user specified assertions by comparing the data structures. If any differences between the two programs are detected, they are reported as well as their location. Using these differences, the user isolates a faulty section of code by repeatedly refining the assertions. Once this section is small enough, traditional debugging techniques can be used to correct the development version. Thus, the debugger provides a quick and effective way of locating problems in the development version.

A relative debugger provides all the functionality of a traditional debugger, including commands for program control, state access and breakpoints. However, several new primitives, not available in conventional debuggers, are required for relative debugging.

Because the reference and development versions of the program are executed concurrently, a relative debugger must be capable of handling two programs at the same time (see Figure 1). If the relative debugger is multilingual, which means being capable of debugging programs in different programming languages, then the reference version and the development version can be written in different programming

languages. An example of relative debugging between a program in Fortran and a program in C is described later in the paper. If the relative debugger is capable of operating in a heterogeneous environment across network, then it can compare program execution on different types of computers. Such a tool is invaluable in transferring programs from one platform to another. In general, as shown in Figure 1, the relative debugger, the reference version and the development version can run on three separate computers of different types.

A relative debugger must be able to compare the state of the reference version with the state of the development version. The comparison is performed by matching the values of variables and data structures from the reference version and the development version. The variables and the data structures to be matched are defined by the user. The relative debugger performs necessary transformations of different internal data representations on different computers or in different languages. When performing comparisons, the debugger must take into account different data types, allowing for such issues as inexact equality in floating point numbers, and differences in dynamic pointer values.

When differences between the execution of both programs are detected by the debugger, they are reported to the user. A number of techniques are possible for reporting the differences. They range from use of printing numbers to data visualization. If there are only a few differences, such as the values of variables or records, then the numeric values of differences are printed out. If differences are numerous, then visualization techniques must be used to present the differences in a meaningful way.

3. Guard: A Relative Debugger

We have implemented a relative debugger, called Guard. The standard set of debugging commands in Guard is expanded with commands for relative debugging. When debugging a single program, Guard is similar to other debuggers. It provides commands to start a process execution, to set breakpoints, to examine the process state, and so on. The main power and novelty of Guard comes from its ability to perform relative debugging.

The following sections describe relative debugging functionality of Guard, a portable debugging interface which is being used to build Guard, and the performance of Guard on two different computing platforms.

3.1. Relative Debugging in Guard

To support the handling of more than one process at the same time, process names are introduced in Guard. The name is assigned when Guard launches a new process or when it attaches to an existing one. Guard can handle an arbitrary number

of processes concurrently, limited only by the restrictions imposed by the operating system.

The comparison of program execution is supported by two different approaches, a *procedural* and a *declarative* approach. We describe the procedural approach first.

The basic command for procedural relative debugging is `compare`:

```
compare <process_1>::<variable_1> <process_2>::<variable_2>
```

`Compare` takes two data structures from two processes as arguments. `Compare` reads both data structures and compares them. The comparison takes into account basic types of both data structures. For example, integers are compared differently from floating point numbers. The user can specify a tolerance value for comparisons. If the difference between the corresponding values from both data structures is greater than the tolerance value, then the difference is reported. The comparison is performed by traversing both data structures and by making a pairwise comparisons between elements. For array comparisons, `compare` interprets array indexes, according to the source language. This enables comparisons of multidimensional arrays between processes whose source languages use different layouts of rows and columns, such as Fortran and C.

To use `compare`, the user must set up breakpoints in both processes, run the programs and, when the programs reach the breakpoints, issue a compare command. This procedure can be tedious and error prone, especially when comparing values within loops. The user must set up breakpoints, run the programs and issue a compare every time a loop is executed. To alleviate the need for this laborious procedure, Guard provides declarative relative debugging. In declarative relative debugging, the user specifies the comparisons to be performed and starts the programs. The handling of breakpoints and the comparisons of data structures are executed automatically by Guard. Comparisons are specified by the `assert` command:

```
assert <process_1>::<breakpoint_1>:<variable_1> =  
      <process_2>::<breakpoint_2>:<variable_2>
```

Each `assert` specifies a pair of breakpoints and their associated data structures. An arbitrary number of assertions can be specified. After all the assertions are set, they are verified by a `verify` command:

```
verify <process_1> <process_2>
```

`Verify` performs the following operations: takes a pair of processes; sets all the breakpoints, specified in the assertions; and starts the programs. When the programs reach their breakpoints, the corresponding assertions are evaluated, and the programs are restarted. `Verify` continuously executes the programs and checks the assertions, until a difference is detected or until a discrepancy in program breakpoints is found.

The procedural and the declarative approach use similar technique to handle differences which are found between programs. In the case of elementary variables, their values and the difference are printed out. In the case of arrays, the maximum difference between the corresponding array elements and the total cumulative difference

between all elements are printed out. In addition for arrays, a rectangular bitmap is displayed to show which array elements are different. White pixels denote values that are the same, black pixels denote values that are different. As demonstrated below, array visualisation is particularly useful at detecting erroneous loop bounds and addressing expressions, because these tend to generate regular patterns on the display. Visualization will also demonstrate processes that slowly drift apart.

Currently, Guard restricts the comparison operations to elementary variables and arrays. The support for more complicated data structures, such as record and dynamic data structures, is planned in the future.

3.2. Implementation of Guard

Guard provides debugging commands, implemented on top of debugging primitives. In a traditional system, these debugging primitives are normally provided by the underlying operating system. This approach poses a number of problems for the implementation of a relative debugger. To a large extent, the debugging interfaces of the operating systems are not compatible between platforms or operating systems. Also, the operating systems usually do not provide direct support for debugging of high level programming languages. Such support is typically implemented as non standard extensions to the interface.

To address these problems we are developing a portable debugging interface, called Dynascope¹⁷. Dynascope provides primitives for building debuggers in distributed and heterogeneous environments. It defines a standard debugging interface which can be implemented at the user level on any number of different operating systems. Further, it can be extended with a direct support for any programming language that is available on the target system.

3.3. Portable Debugging Interface

The key to Dynascope's portability is the provision of high level *debugging* primitives for performing program monitoring and controlling. The program monitoring primitives are used to obtain information about the program execution without changing the semantics of the program. The program controlling primitives are used to modify the program state with the purpose of changing program's future behavior.

Dynascope provides primitives for execution control, state access, breakpoints, tracing, and dynamic linking and loading. A detailed description of the interface can be found in Sosič¹⁷. Guard uses the following Dynascope primitives to provide commands for relative debugging: execution control, state access and breakpoints.

The primitives in the execution control group enable Guard to attach to or detach from an arbitrary process. Guard can attach to a process on a different computer of the same or of a different type, as long as computers are connected by a network. The

network can range from a local area network to Internet. From the Guard point of view, `attach` is the only Dynascope primitive that distinguishes between debugging a program on the local machine and debugging a program across a network. Dynascope provides the necessary communication framework, so that the rest of the Dynascope primitives are identical for local or remote debugging. Additional primitives in this group can start or stop the process.

Primitives in the state access group allow Guard to obtain the state of a process. Any part of the user accessible process's state can be read by Guard, including the process's address space and internal registers.

Primitives in the breakpoint group support the setting and deletion of breakpoints. Two primitives, which provide a correspondence between the source code and the machine code, assist in dealing with breakpoints. One primitive returns symbol values from the user's program, whilst the second matches lines from the source code with instruction addresses.

The debugging interface is implemented using remote procedure calls. Guard acts as a client that issues debugging requests through remote calls to a debugging server. The debugging server services debugging requests and returns results to Guard. Each process has a separate debugging server, which is part of the process's code. The debugging server shares control with the corresponding user program. When a debugging request is received, the execution of the user program is stopped and the server starts servicing the request. When the server finishes, it exits and returns the control to the user program. This organisation has a number of advantages. First, if the server is not activated, it does not slow down the user program. Second, the server is not visible to the user program, because there are no direct calls from the user program to the debugging server. Therefore, Guard can be used on any user program that is linked with the debugging server.

Currently, debugging servers are implemented on two platforms, on SunOS and Nextstep. Using these servers, Guard and processes can be arbitrarily mixed across these two platforms. A detailed description of Dynascope implementations can be found in Sosic ¹⁶.

4. The Shallow Water Equations: A Sample application of Guard

We have performed an evaluation of Guard by debugging a version of the Shallow program. Shallow is a simple numerical model of the shallow water equations ^{13,6}.

The shallow water equations describe the motion of an incompressible fluid with a free surface, with the constraint that the horizontal scales of motion are much larger than the vertical. The equations are a favoured choice for experiments with various model structures and numerical schemes, and thus must be ported to many different computing platforms. Although they use a very simple representation of the atmosphere, they do include the two types of horizontal wave motion important in

more realistic global climate models, gravity waves and Rossby waves.

As part of a separate research project we ported the shallow model to a number of different platforms ¹. These included different sequential machines, different programming languages, shared memory and distributed memory parallel computers. The process was quite difficult in spite of the small size of the programs, because the code involved the manipulation of large floating point data structures. In order to compare the execution of any version of Shallow to the reference program we were forced to output the state of the program to a file at each time step, and then to compare the state traces. This process was extremely time consuming and error prone. Without a relative debugger we were unable to determine easily why a program was misbehaving.

We have decided to use the Shallow program to evaluate the effectiveness of Guard. We have compared the reference version, written in Fortran-77, with a development version written in C. In our experiments, both the reference code and the debugged code were actually executed on the same computer system. However, Dynascope provides a debugging interface which isolates the application from the underlying execution platforms. Thus, we could equally have performed the experiments across networked systems. In the following sections we demonstrate the use of Guard in detecting a number of errors that were introduced when the Shallow program was ported from Fortran to C. This comparison demonstrates that Guard can operate across different programming languages which use different internal representations for data structures.

4.1. Experiment 1

Whilst the Fortran and C programs implement the same basic algorithm, they are structured differently. The Fortran code implements the periodic boundary condition of the equations by computing values for all grid cells, and then copying the last column and row back over the first row and column. This structure was used to improve the vectorisation properties of the code. The C code, on the other hand, uses modulo arithmetic on the array indexes to force the last row and column index values to equal 0. Since it was never meant to execute on a vector processor, we were more concerned with elegance than efficiency.

This difference means that the arrays in the Fortran code must be one cell larger than the C version. In our initial port, we accidentally set the array sizes to be the same in both codes, and the two programs generated different solutions. Guard was used to determine that the arrays containing the pressure values were different, which in turn was caused by different values in one of the initialised data structures (called psi). Figure 2 shows a visualisation produced by Guard, after detecting the differences in data structures. Guard displays a black pixel in the row and column position of the array at which the two data values differ. Clearly, every value of the

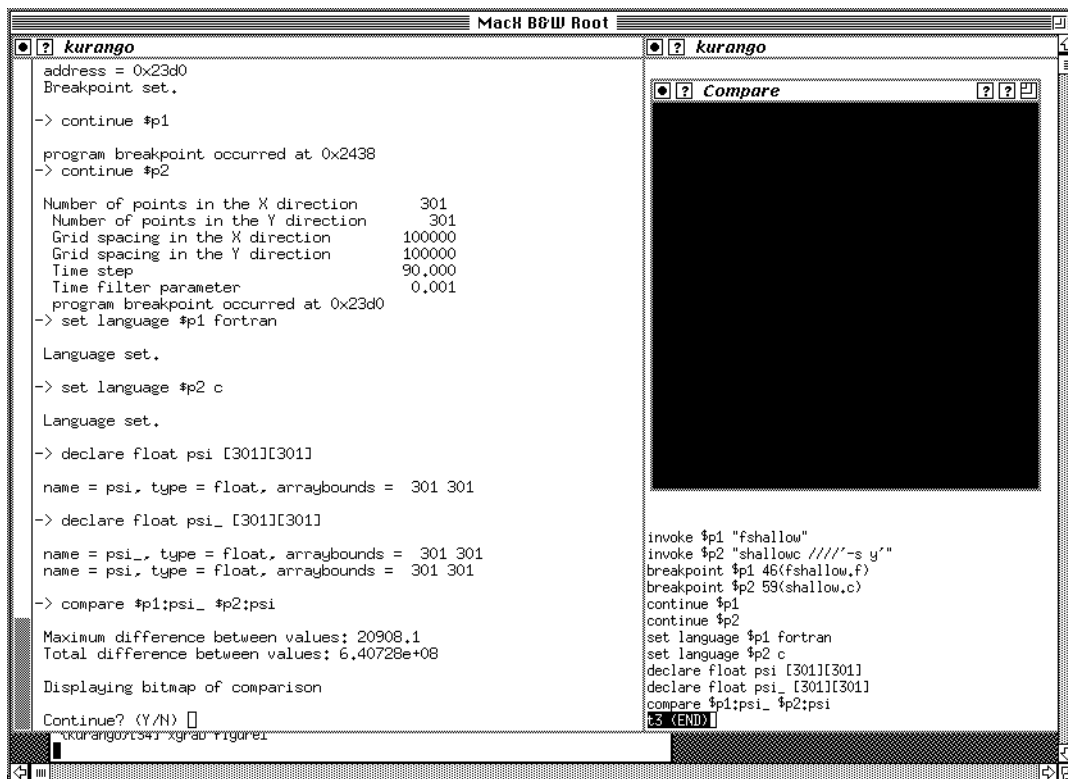


Figure 2: Differences in data structure caused by wrong array bounds

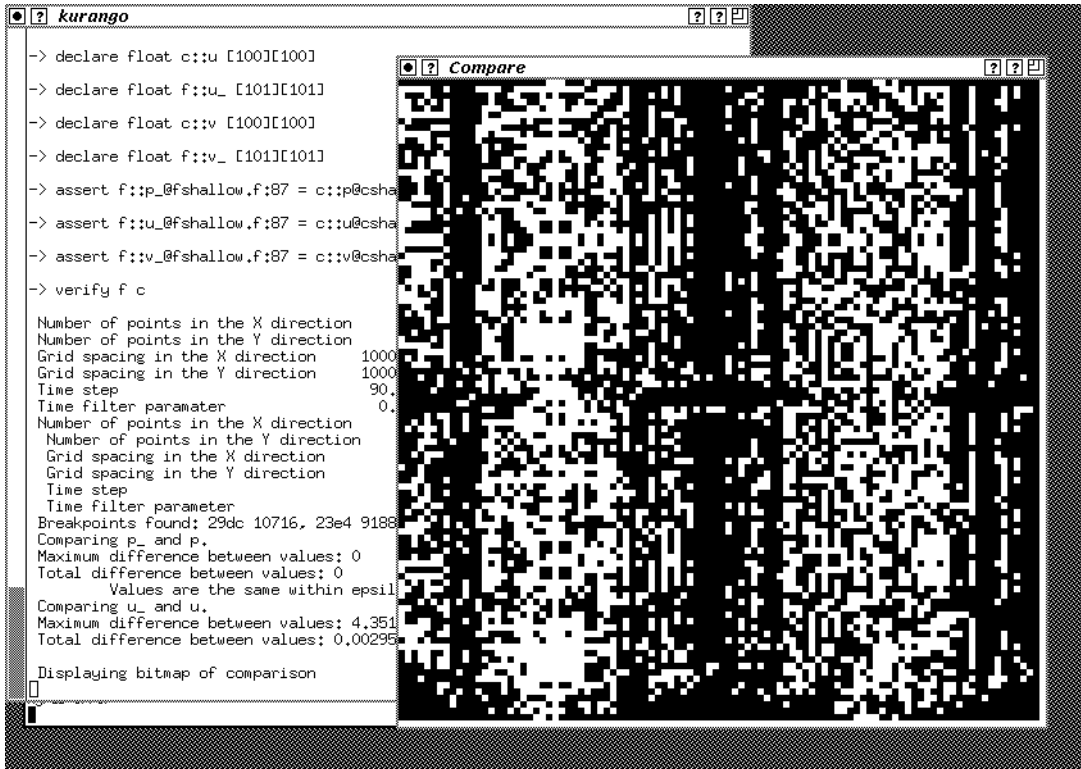


Figure 3: Incorrect boolean expression for detecting first step

array is incorrect. Further, the maximum difference is very large, and has a value of 20908.1. Once this visualisation was displayed it took little time to determine that the array bounds were incorrect in the C code, and the problem was rectified.

4.2. Experiment 2

The implementation of the shallow water equations applies a time filter to the numeric integration scheme, which means that the time step is altered after the first iteration. When we executed the two programs they produced the same output after the first iteration, however, the values began to drift in subsequent iterations. The Guard display, shown in Figure 3, demonstrates that the data structures are the same after the first iteration, but that they contain different values in the second iteration. A more detailed examination of the code revealed that this was caused by an incorrect test of the flag which marked the first time around the loop. Removing an incorrect inversion on the predicate removed this error.

4.3. Experiment 3

The third error was caused by an incorrect loop bound. The visualization pro-

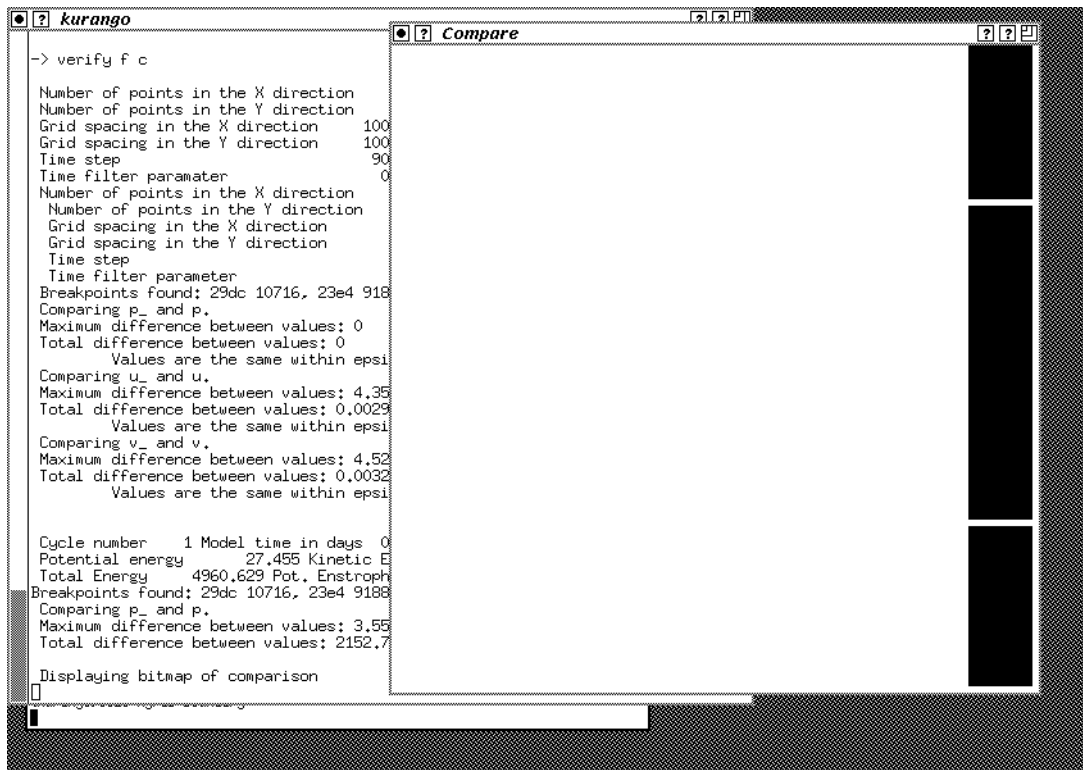


Figure 4: Incorrect loop bound

duced by Guard clearly shows the effect of the column index terminating too soon (see Figure 4). This is shown by the a black stripe on the right side of the display, which indicates that the two arrays differ in a number of columns. The Guard visualisation scheme it possible to detect array bound errors easily because they appear as regular patterns on the visualisation.

5. Future Directions

Relative debugging has a wide range of applications. A major use of relative debugging can be found in the reengineering and maintenance of computer programs and other activities that modify software. Relative debugging can be applied whenever an existing program is changed. These changes can involve a different programming language, a different execution model, such as a parallel or distributed version of a sequential program, a different type of the computer, optimisations of algorithms, or the addition of new features.

Because relative debugging is a new concept, our activities concentrate on the investigation of its potential. Several questions arose while using Guard, our relative debugger. What is the best way to specify assertions and assure their consistency? How should results of comparisons be displayed? What is the most efficient way to

compare dynamic data structures? We have addressed some of these issues in this paper. For a better model of relative debugging and the associated trade-offs, we need to gain more experience with practical applications.

Another area of research is the development of Dynascope, our portable debugging interface. Guard and Dynascope are being developed in parallel, with Guard providing the guidance for new primitives and trade-offs in Dynascope. Dynascope is being ported to new computing platforms which will automatically expand the number of computer systems supported by Guard.

An area where we expect large benefits from relative debugging is the development of parallel programs. Due to a large number of concurrently executing control threads, parallel programs are very hard to develop and debug. However, in many cases, parallel programs have a corresponding sequential version which is correct. With the help of a relative debugger, this sequential version can be used to locate problems with parallel programs much faster than otherwise possible. Parallel programs bring new issues into relative debugging, such as synchronisation of processes and a distributed nature of data. We plan to extend Guard with features to support parallel programming.

6. Acknowledgements

This work has been sponsored in part by the Australian Research Council. The authors wish to acknowledge the work of Ms. Lisa Bell, who has performed most of the programming necessary to implement Guard.

7. References

1. D. A. Abramson, M. Dix, and P. Whiting. A study of the shallow water equations on various parallel architectures. In *14th Australian Computer Science Conference, Sydney, 1991*.
2. Z. Aral, I. Gertner, and G. Schaffer. Efficient debugging primitives for multiprocessors. In *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 87–95. ACM, 1989.
3. I. J. P. Elshoff. A distributed debugger for amoeba. In *Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 1–10. ACM, 1988.
4. N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings Supercomputing-93, Portland, Oregon*, pages 462–471. IEEE, 1993.
5. J. H. Griffin, H. J. Wasserman, and L. P. McGavran. A debugger for parallel processes. *Software-Practice and Experience*, 18(12):1179–1190, December

- 1988.
6. G. J. Haltiner and R. T. Williams. *Numerical Prediction and Dynamic Meteorology*. Wiley, New York, 1980.
 7. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.
 8. C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.
 9. B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 135–144. ACM, 1988.
 10. T. G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.
 11. B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.
 12. R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Software-Practice and Experience*, 21(2):209–229, February 1991.
 13. J. Pedlosky. *Geophysical Fluid Dynamics*. Springer-Verlag, Berlin, 1979.
 14. E. Satterthwaite. Debugging tools for high level languages. *Software-Practice and Experience*, 2(3):197–217, July-September 1972.
 15. T. Shimomura and S. Isoda. Linked-list visualization for debugging. *IEEE Software*, 8(3):44–51, May 1991.
 16. R. Sosič. Implementation of directing for compiled programs. Technical Report CIT-1994-7, School of Computing and Information Technology, Griffith University, Brisbane, 1994.
 17. R. Sosič. A procedural interface for program directing. Technical Report CIT-1994-5, School of Computing and Information Technology, Griffith University, Brisbane, 1994.
 18. T. G. Stockham, Jr. Some methods of graphical debugging. In *Proc. IBM Scientific Computing Symposium on Man-Machine Communication*, pages 57–71, 1965.
 19. P. T. Zellweger. Interactive source-level debugging of optimized programs. Technical Report CSL-84-5, Xerox PARC, 1984.