Guard: A Relative Debugger

Rok Sosič and David Abramson
{sosic, davida}@cit.gu.edu.au
School of Computing and Information Technology
Griffith University
Brisbane, QLD 4111
Australia

(to appear in Software - Practice and Experience)

Abstract

A significant amount of software development is evolutionary, involving the modification of already existing programs. To a large extent, the modified programs produce the same results as the original program. This similarity between the original program and the development program is utilized by *relative debugging*.

Relative debugging is a new concept that enables the user to compare the execution of two programs by specifying the expected correspondences between their states. A relative debugger concurrently executes the programs, verifies the correspondences, and reports any differences found. We describe our novel debugger, called Guard, and its relative debugging capabilities. Guard is implemented by using our library of debugging routines, called Dynascope, which provides debugging primitives in heterogeneous networked environments.

To demonstrate the capacity of Guard for debugging in heterogeneous environments, we describe an experiment in which the execution of two programs is compared across Internet. The programs are written in different programming languages and executing on different computing platforms.

Keywords: Debugging, Programming Tools, Program Development, Relative Debugging.

1 Introduction

A large portion of software development is based on the evolution of computer programs. Instead of developing new programs, existing programs are modified. Some evolutionary software development activities are: redesign of programs in order to improve their performance or add new functionality; porting of programs to new computing platforms; and development of parallel programs from sequential code. These activities are similar in that they do not change the basic program functionality. The problems of debugging software under these conditions are significant and represent a major cost to the software industry. Since the original program version and the development version perform essentially the same task, it is possible to use the original program version as a specification for the development version. The execution of the development version can be compared with the execution of the original version. Any differences between the programs can indicate an error in the development version. The usage of a trusted original version of the program to test the development version can significantly speed up software testing by automating large parts of the testing process.

One of the existing approaches is to compare the programs' source code [14]. Although it is capable of handling a wide range of programming constructs, this approach is inherently limited. In general, the exact program execution cannot be predicted from the source code. A more powerful approach is to use the original version as an *executable* specification for the development version. This approach is provided by our novel debugging technique, called *relative debugging* [2].

In relative debugging, the debugger controls the execution of two programs. The programs are executed concurrently. The execution is interrupted at specified points and the debugger compares sections of program states. If the sections differ, the differences are reported. Otherwise, the execution continues. The comparison points and the sections to be compared are specified by the user. Automatic methods to derive these points and the sections are not addressed here.

Although the significance of comparing the execution of two programs has been recognized for a long time, there are almost no supporting tools. In its most elementary form, the comparison is performed manually by running two programs side by side, possibly under control of two separate debuggers. Each program requires its own terminal or a window. The user manages the execution of programs and visually compares the values of data structures that are printed on the screen. This approach is extremely tedious and error prone. It provides only limited insight, especially if the programs contain large data structures or execute for a long time. The approach prevents automatic methods for visualizing differences, which are capable of exposing subtle errors.

Some of the limitations of the manual approach can be avoided if parts of the program's state are saved to a file at regular intervals [12]. This approach requires modifications to the program source code to insert state saving routines. The files, produced by the two programs, are compared using a file comparison utility, such as `diff` on Unix. One limitation of this technique is that the required disk space grows linearly with the program execution time. Another limitation is that file comparison utilities usually compare files on a character by character basis without taking into account type information. This approach can present a problem with floating point numbers, which use an inexact representation.

Debugging technology is ideally suited for building a tool to support the process of software evolution by providing necessary primitives for comparing program execution. Debuggers can control the program execution, access the program's state and the relevant type information, and do not require source code [6, 8, 18, 25, 26, 27, 34, 35]. These capabilities of debuggers can overcome the limitations of the manual and the file comparison approaches.

This paper describes our relative debugger, called Guard [1]. In addition to ordinary debugging commands, Guard provides new commands for relative debugging. We discuss some conceptual issues in relative debugging in Section 2. These issues include the specification of comparison points, control of processes, debugging approaches, and the comparison of data structures. Novel capabilities of Guard, specific to relative debugging, are described in Section 3. These capabilities include: process invocation and control; imperative and declarative debugging; the display of differences; automatic generation of assertions; and execution traces. The implementation of

[1] patent pending

Guard is described in Section 4. By using our library of debugging primitives, called Dynascope [31], Guard is capable of operating in multi lingual, heterogeneous networked environments. We demonstrate this capability by running Guard on a Silicon Graphics computer and by comparing the execution of two programs written in two different programming languages, C and Fortran, and running on two different platforms, Sun and NeXT. We conclude by describing some future possibilities for relative debugging.

2 Issues in Relative Debugging

The use of relative debugging is simple. The programmers identify important data structures in the development program. Next, they establish the corresponding data structures in the original program. Finally, they specify locations in both programs at which the correspondence between the data structures is expected. Both programs are run under the control of a relative debugger. If no differences are found, then the programs produce expected results and the development version is assumed to be correct. If the debugger finds any differences, these are reported. The programmers can repeat the process of program comparison by refining locations at which the data structures are compared or by specifying additional data structures to be compared. The process continues until a faulty section of the code is identified. This approach is capable of finding errors in a very short time. For example, we were able to locate an error in a Fortran program with 15,000 lines of code in just three iterations through the program comparison process with a total of eight assertions [2]. In another experiment, a person with no prior knowledge of a 30,000 line program was able to identify divergence between two versions of the program in less than an hour [1]. It is interesting that the two versions were assumed to be functionally equivalent by the authors of the program.

Although relative debugging is conceptually simple, a relative debugger involves significant challenges, demanding new approaches and techniques. The debugger must be able to control and synchronize concurrent execution of two programs. This control requires that processes are interrupted at predetermined points. After the processes are interrupted, the debugger accesses variables from both programs and compares their values. These comparisons can be nontrivial. For example, dynamic data structures with pointers are laid out differently for each program. In addition, the inexact nature of the floating point arithmetic must be taken into account. The programs can be written in different languages, which use different representations of data structures. For example, an array can be stored in a row major or in a column major form or even laid out in a more complex arrangement to minimize communication effects. The debugger must be able to establish a correspondence between different representations of data structures. After a comparison is performed, the debugger must report its results to the user. The method for reporting the results can be textual or graphical, depending on the number of differences.

The following parts of this section describe some of the issues in relative debugging in detail. The issues addressed are the specification of comparison points, control of processes, debugging approaches, and the comparison of data structures.

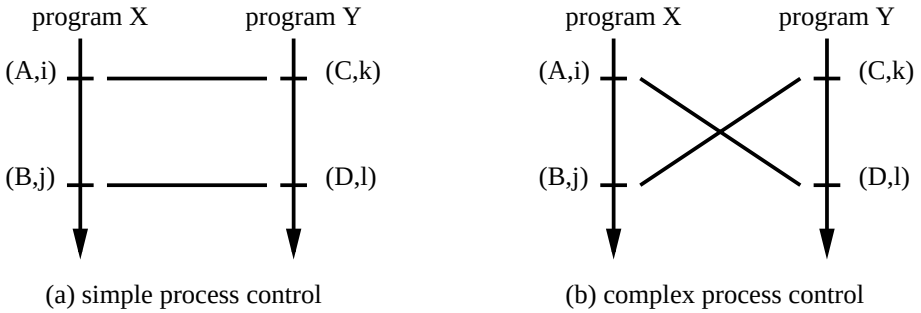

Figure 1: Control of Processes

2.1 Specifying Comparisons

The relative debugger verifies that two processes are producing similar results. The debugger accomplishes this verification by continually interrupting the processes and comparing some of their variables. The interruption points and variables to be compared are specified by the user.

We call interruption points with the corresponding variables *containers*. A container describes a location, at which the program is interrupted, and a variable, whose value is significant at that point. The location can be specified by a line in the source code or by an address. The variable description consists of the access information, which can be either the variable name or its address, and the type. The type can be supplied by the user or extracted from the debugging information produced by the compiler. For each container, the location must be within the valid scope for the variable. The same location or the same variable can be specified in more than one container.

In relative debugging, the user states the relationships between containers from different programs. The simplest relationships might require that variables contain the same values. More complex relationships might compare only portions of variables. The most generic relationships would allow arbitrarily complex mappings for establishing the correspondence between variables.

2.2 Control of Processes

After the containers are defined and their relationships are specified, the user starts a debugging session. During the session, the relative debugger initiates the two programs and sets breakpoints at the union of all container locations. The execution of both programs is resumed and the debugger waits for both programs to reach breakpoints. When the breakpoints are reached, the containers with these breakpoints as locations are identified. The debugger copies the variables from programs to containers and verifies that the specified relationships between containers are satisfied. If any of the relationships is violated, then the debugger reports an error. Otherwise, programs resume the execution until a new set of breakpoints is detected.

The control of processes in relative debugging is nontrivial. A fundamental issue is the relationship between container comparisons and breakpoints. If the order in which breakpoints are encountered by the programs corresponds to the order in which containers are to be compared, then control of processes is straightforward. This case is illustrated in Figure 1(a). Two containers are specified for program `X`. The first container `(A,i)` has location `A` and variable `i`.

The second container has location `B` and variable `j`. Containers `(C,k)` and `(D,l)` are specified for program `Y`. Containers to be compared are `(A,i)` with `(C,k)` and `(B,j)` with `(D,l)`. After the programs encounter breakpoints `A` and `C`, the container comparison can be performed and the programs can resume their execution.

An example of complex process control is shown in Figure 1(b). In this case, containers to be compared do not match the order in which breakpoints are encountered. Such cases might arise, whenever the control structure of the development program is different from the original program. Program `X` will reach breakpoint `A` and program `Y` will reach breakpoint `C`. At this point, no container comparisons could be performed, because the matching containers are not yet available. If the program execution is resumed, then containers `(A,i)` and `(C,k)` could be overwritten with some other value by the time programs reach breakpoints `B` and `D`. A solution is to keep the values of variables `i` and `k`, until they are consumed by a comparison. As a result, each container must maintain a queue of variable values. Whenever a breakpoint is encountered, the value of the corresponding variable is placed in the container's queue. When both containers in a comparison have at least one value in their queues, the comparison is performed and values are removed from the queues. This approach supports complex control of processes, because containers can be compared in any order. In practice, a limit is put on the queue length, which prevents a potential memory overflow by one of the processes. In the worst case, which is rare in practice, the storage requirements for containers might grow linearly with the execution time.

2.3 Debugging Approaches

Relative debugging involves executing the programs, interrupting them at relevant locations and performing the comparisons of corresponding data structures.

In the simplest approach, these operations can be issued manually. This approach is similar to traditional debugging techniques. The user implants breakpoints, which determine locations of interest. After the program hits a breakpoint, it is interrupted and its state is inspected. Program execution and state inspection are repeated, until an error is detected.

Relative debugging can be performed in a similar way by manually controlling program execution. Before the execution, the user implants breakpoints in both programs, the reference version and the development version. The breakpoints are placed at locations, at which some variables in both programs are expected to contain similar values. After the programs encounter breakpoints during the execution, breakpoint positions are verified for correspondence and the relevant variables are compared. If the breakpoints occur at unexpected locations, or the data structures differ, an error is found. Otherwise, the programs resume their execution and the process is repeated.

This manual control of programs is called an *imperative* approach, because the order in which breakpoints and the comparisons are issued is controlled explicitly by the user. However, the imperative approach is unsuitable for complex programs. Because several commands must be issued for each comparison performed, it can become time consuming and tedious. For example, if variables to be compared are in a loop, the user must provide commands to implant breakpoints, compare variables, and resume the execution of programs for each loop iteration. If the loop is repeated many times, then this manual approach becomes impractical. One solution

```
function Compare (A,B) : { different, equal }
begin
    if not EqualTypes(A,B) then return(different);
    if TypeSimple(A) then return(CompareSimple(A,B));
    if TypeComposite(A) then return(CompareComposite(A,B));
    if TypePointer(A) then return(ComparePointer(A,B));
end;
```

Figure 2: Comparing Generic Variables A and B

could be to place the commands in a script file, one sequence of commands for each loop iteration. A problem with this solution is that it is capable of handling only loops with a predetermined number of iterations.

An alternative to the imperative approach is a *declarative* approach. In the declarative approach, the user does not issue debugging commands explicitly for each comparison. Instead, the user specifies *assertions*, which establish a relationship between containers in two programs. After the assertions are set, a single debugging command is issued by the user, regardless of the number of containers and assertions. The command automatically sets all the necessary breakpoints, controls the execution of processes, and performs relevant variable comparisons, using the process control approach described in Section 2.2. If an assertion is not satisfied during the execution, the debugger reports relevant data structures and exits the declarative mode. The declarative approach enables the user to specify the correspondence between programs in a natural and efficient way, which greatly increases the power of relative debuggers.

The imperative and declarative approaches both require the comparison of variables. Although one approach is manual and one automatic, they use the same comparison method. The method is described in the next section.

2.4 Comparing Data Structures

The relative debugger compares variables from different programs by copying their values to its own address space and performing the comparisons. The comparison is done in a recursive fashion. Data structures are decomposed into basic elements, which are compared directly. Pointers are compared indirectly by comparing their references. If any two basic elements are found different, then the variables are declared different. The details are described below.

For the comparison purposes, types of data structures are defined as being either simple, composite or pointers. Simple types are basic types, such as integer, floating point and character, from which all other types are constructed. These types are usually directly supported by the underlying hardware. Composite types, which can be arrays or records, contain one or more elements. In the case of arrays, the elements have the same type. They are accessed by index. In the case of records, the elements can have different types. They are accessed by their names. Pointers contain addresses of other elements.

To compare two variables, the relative debugger uses function `Compare` (see Figure 2). `Compare` takes two input parameters, `A` and `B`. The parameters specify the variables to be com-

```
function CompareSimple (A,B) : { different, equal }
begin
    if TypeInteger(A) then return(CompareInt(A,B));
    if TypeCharacter(A) then return(CompareChar(A,B));
    if TypeFloat(A) then return(CompareFloat(A,B));
end;
```

Figure 3: Comparing Simple Variables A and B

pared, their types, and the programs in which they reside. `A` and `B` can be in two different programs. It is assumed that the programs containing variables are interrupted and that the debugger has access to type information and data space for both programs. `Compare` returns *equal*, if the values of `A` and `B` are the same. Otherwise, it returns *different*.

`Compare` first verifies that `A` and `B` have the same type. Next, it calls `CompareSimple`, `CompareComposite` or `ComparePointer`, depending on the type of `A` and `B`. These functions compare simple, composite or pointer types, respectively.

`CompareSimple` is shown in Figure 3. It determines the type and calls the corresponding comparison procedure. Comparison procedures access the variables, copy their values to the relative debugger, and perform the comparison. Because integer and character arithmetic is exact, `CompareInt` and `CompareChar` are straightforward in principle. In practice, however, the debugger must take into account different character sets and different integer representations, such as different byte ordering schemes, different sizes of integers in heterogeneous systems, and different representations of negative numbers. Before a comparison can be performed, values must be converted to a canonical form, understood by the debugger. In addition to different representations, floating point comparisons, performed by `CompareFloat`, have to take into account the inexact nature of floating point arithmetic. Mathematically equivalent, but computationally different sequences of floating point operations usually produce different results. This inequality is accommodated by a user specified tolerance value. If the difference is less than the tolerance, then the numbers are considered equal, otherwise they are different.

`CompareComposite` performs a comparison of composite variables. It simply checks whether the variables are either arrays or records. In the case of arrays, these are traversed from the first element to the last and the comparisons between the corresponding elements are performed. Indexes, which are used to access arrays, represent a natural way to establish a correspondence between elements of two arrays. In the case of records, the correspondence might be nontrivial. The problem is similar to the problem of determining type equivalence in compilers [4]. One solution is to compare record fields in their order of occurrence by default and to allow users to define other correspondence orders.

Pointer comparisons are more complicated than comparisons of other types. Pointer values between programs will in general differ in value because of different heap management techniques. Other data structures are compared by comparing the values of the corresponding elements. With pointers, their references are compared instead. A minimal or a maximal approach can be taken for comparing pointers. In the minimal approach, only the immediate pointer references

```
function ComparePointer (A,B) : { different, equal }
begin
    if InTable(A) or InTable(B) then
        if InTable(A) and InTable(B) then return(equal)
        else return(different);
    Add(A); Add(B);
    return(Compare(*A,*B));
end;
```

Figure 4: Comparing Dynamic Data Structures A and B

are compared. In the maximal approach, the entire data structure, to which the pointer is pointing, is traversed and the corresponding elements are compared. We describe here the more general, maximal approach.

`ComparePointer` is shown in Figure 4. To prevent an infinite loop while traversing pointers connected in a cycle, a table of already visited pointers is maintained. This table is initialized, before the pointer comparison starts. `ComparePointer` uses function `InTable` to check if a pointer is in the table and procedure `Add` to add a pointer to the table. The reference of pointer `A` is denoted by *A. `ComparePointer` first checks, if the pointers have been visited before. If both of them have been visited, then their references are equal. If only one of the pointers have been visited, then the dynamic data structures are different. If none of the pointers have been visited, then the pointers are added to the table and their references are compared. This approach verifies that elements in both data structures are connected in the same way and that they contain same values.

3 Relative Debugging in Guard

This section describes facilities for relative debugging in Guard, our relative debugger. Guard also supports traditional debugging commands. Because these commands are commonly found in most debuggers, they are not discussed in any detail. Facilities for relative debugging provide process control; imperative and declarative debugging for comparing program execution; methods for displaying differences; support for automatic generation of assertions; and execution traces. The most important of these features are described below.

3.1 Process Invocation and Control

To provide the control of multiple processes, Guard supports process names. Each process has a name, which is used to identify the process. The name is assigned by the user at the process activation time. There are two methods to activate a process.

The user can attach to an existing process or invoke a new process:

```
attach <process_name> <process_id> <computer_name> <user_name>
invoke <process_name> <command_line> <computer_name> <user_name>
```

<process_name> is a user assigned logical name, which identifies the process in subsequent commands. The target process is specified either by its process identification number or a command line to start the process. <computer_name> and <user_name> specify the execution environment. <computer_name> specifies the computer, <user_name> specifies the user name. attach and invoke are the only two commands that require the name of the target computer. Other commands require only logical process names. The process can execute on the same machine as the debugger or on a remote machine, connected to the debugger's machine by a network. The only features of debugging commands that distinguish remote debugging from local debugging are machine and user parameters in invoke and attach commands. If these two parameters are empty, programs are invoked on the local machine, otherwise they are invoked on remote machines.

3.2 Imperative Debugging

Guard implements both imperative and declarative relative debugging as discussed in Section 2.3. The elementary imperative command is compare. It can be issued when the processes are interrupted, usually after both processes encountered a breakpoint. Compare takes two variables, possibly from two different processes, and reports any differences in their values:

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

During the execution of compare, the relative debugger copies the values of both compared variables from the programs into its internal buffers and performs the comparison. If the two values differ, Guard reports the results by printing a textual report or by visualizing differences directly or through an external visualizing program. The comparison takes into account type declarations in the corresponding source programs. If present, the type information can be extracted directly from the executable files. Otherwise, it can be supplied by the user.

The existing version of Guard supports simple types and arrays, because we have concentrated on numerical programs, which contain only simple types and arrays. The compare command is being extended to handle more complex types, such as records and dynamically allocated data structures, using the approach described in Section 2.4.

The inequality of floating point numbers is handled by a user specified threshold. The threshold can be either global or specified for a particular comparison. Furthermore, the threshold can be relative or absolute. Our experience in using Guard shows that both methods of calculating the threshold are necessary.

In comparing arrays, additional complexity is provided by different array representations in different languages. For example, multidimensional arrays can be laid out in a row major or a column major order. Also, the first element can start with a different index: with 0 in C, with 1 in Fortran, and with a user specified value in Pascal. To indicate different array layouts, the user can associate one of the predefined *language types* with each process. Guard then uses this information in determining array layouts.

It is often desirable to compare only limited array sections. This partial array comparison is supported by array slices, which span only a partial array range. Such slices can be specified by the user in the compare command by giving the comparison range. Additional array transformations, supported by Guard, include index permutations and pointers to arrays.

```
 1.            # start process "f": the program in Fortran
 2.   invoke f fshallow machine1.cit.gu.edu.au user1
 3.            # start process "c": the program in C
 4.   invoke c cshallow machine2.harvard.edu user2
 5.            # specify language for processes "f" and "c"
 6.   set language f fortran
 7.   set language c c
 8.            # specify the dimensions of array P in the Fortran program
 9.   declare float f::p [301][301]
10.            # specify the dimensions of array pr in the C program
11.   declare float c::pr[300][300]
12.            # specify difference threshold for floating point numbers
13.   eps absolute float .01
14.            # set breakpoints in "f" and "c"
15.   breakpoint f::fshallow.f:87
16.   breakpoint c::cshallow.c:63
17.            # continue with the execution of "f" and "c"
18.   continue f
19.   continue c
20.            # breakpoints were encountered, compare the array values
21.   compare f::p[1..300][1..300] c::pr[0..299][0..299]
      ...
      ...       # lines 18.-21. must be repeatedly typed in by the user
      ...       # for each loop iteration
      ...
```

Figure 5: Imperative Debugging

The comparison of arrays is illustrated by the following example. Structures to be compared are array `P` in a Fortran program and array `pr` in a C program. `P` is defined in Fortran as:

```
DIMENSION P(301,301)
```

`pr` is defined in C as:

```
float pr[300][300];
```

Array dimensions in Fortran are larger by one because the code is optimized for pipelining in vector processors. A complete script of Guard commands is shown in Figure 5. The script compares the corresponding sections of arrays `P` in Fortran and `pr` in C. The last line of the script contains a `compare` command. Array layouts are determined by the `set language` commands. Slices are determined by specifying the processes, variables and ranges to be compared, `f::p[1..300][1..300]` and `c::pr[0..299][0..299]`. From the specification of array layouts and slices, the positions of individual array elements are calculated. The elements are then accessed and compared.

With the `compare` command, the notion of a container does not exist. All the program control and comparisons are managed explicitly by the user, which can be tedious in some cases. A common place to compare variables is within a loop. The `compare` command will perform

```
      ...
      ...       # same as lines 1.-13. in imperative debugging
      ...
14.             # make an assertion
15.   assert f::p [1..300][1..300]@fshallow.f:87 =
16.          c::pr[0..299][0..299]@cshallow.c:63
17.             # verify the assertions for processes "f" and "c"
18.   verify f c
```

Figure 6: Declarative Debugging

only a single comparison, so the user must continually issue commands for each loop iteration. Imperative debugging is thus suited only for simple programs. For more complicated programs, declarative debugging provides a significantly better alternative.

3.3 Declarative Debugging

Declarative commands, which are based on containers, significantly simplify relative debugging. These commands are `assert` and `verify`. `assert` provides an assertion, specifying containers to be compared:

```
assert <process_1>::<container_1> = <process_2>::<container_2>
```

Each container is composed of a variable to be compared and a breakpoint location. An arbitrary number of assertions can be specified by the user. After all the assertions are set, they are verified by a single `verify` command. Without any further user intervention, the `verify` command sets all the breakpoints, manages the process execution, and performs the relevant comparisons when breakpoints are encountered. The same method for comparing variables is used by `verify` and `compare`. The `verify` command terminates and returns control to the user, if: the pair of breakpoints encountered does not have at least one corresponding assertion; comparisons detect differences; or programs terminate.

Our existing version of Guard supports only simple control of processes. When the processes encounter breakpoints, an error is reported if only one of the containers in an assertion is available. If both containers are available, then the comparison is performed and the process execution is resumed, if no differences are found. This simple control of processes is sufficient in a majority of cases.

The example in Figure 5, can be rewritten with assertions. Commands for imperative debugging in lines `14.-21.` can be replaced with commands for declarative debugging in Figure 6. A significant advantage of relative debugging is that a single `verify` command is required to perform the comparison every time process `f` reaches line 87 in file `fshallow.f` and process `c` reaches line 63 in file `cshallow.c`. If imperative debugging is used instead, several commands must be issued explicitly for each comparison performed.

3.4 Displaying the Differences

If programs satisfy assertions, then no differences are reported by Guard. However, if Guard finds any differences, these are reported to the user. The report of differences can be textual or graphical.

Text is used to report the differences between simple variables. A similar approach is used for arrays. Assertions with arrays involve comparisons of corresponding elements. If any differences are detected, a list of pairs of array elements is printed, together with their indexes and differences. However, such a report is not appropriate for all programs and can be prohibitively long for large arrays. The user can select to print out only some of the differences or switch off the printing completely. The difference report can be also redirected to a file. This redirection capability is especially useful, when differences are post processed with a visualization package, capable of performing advanced multidimensional visualizations of large quantities of data. It is possible to combine several consecutive visualizations of differences into an animation. Animations of differences proved extremely helpful in analyzing numerical computer models, where slight algorithm modifications can significantly alter the result. A case study of using Guard on a real program, a large-scale weather model, is described elsewhere [1].

Guard itself provides a simple, but powerful two dimensional visualization of array differences. Arrays are represented by a rectangle on the screen, so that each screen location corresponds to one pair of array elements. If the elements from the two arrays are equal, the corresponding location on the screen is white, otherwise, the location is black. The mapping of a two dimensional array on a rectangle on the screen is obvious. If an array is only one dimensional, then it can be broken into several intervals to form a rectangle on the screen. If an array has more than two dimensions, then redundant dimensions can be collapsed into two dimensions.

Guard visualization utilizes the human capabilities for pattern recognition to give an effective way of detecting and diagnosing anomalies in the program behavior. An example of a graphical display of differences between two programs is shown in Figure 7. The two programs being compared are written in Fortran and C and calculate shallow water equations [3]. The largely black section at the bottom demonstrates that the arrays almost completely differ in that region. From the picture, it was quickly determined that one of the loops terminated prematurely, so a large portion of the array had not been updated. The elaborate pattern in the top region shows differences in floating point values between the Fortran and the C version. Although both programs compute correct values in the top region, the differences are produced by inexact floating point arithmetic. The pattern at the top disappears, if the threshold value is increased.

To assist with the threshold value, an array comparison always produces a short summary of differences. The summary contains two numbers, the maximum difference and the total difference. The maximum difference is the largest difference found across all the element comparisons performed. The total difference is the sum of all differences between the corresponding pairs of elements. By observing these numbers, the user can obtain a quick estimate of the differences without analyzing detailed numbers.

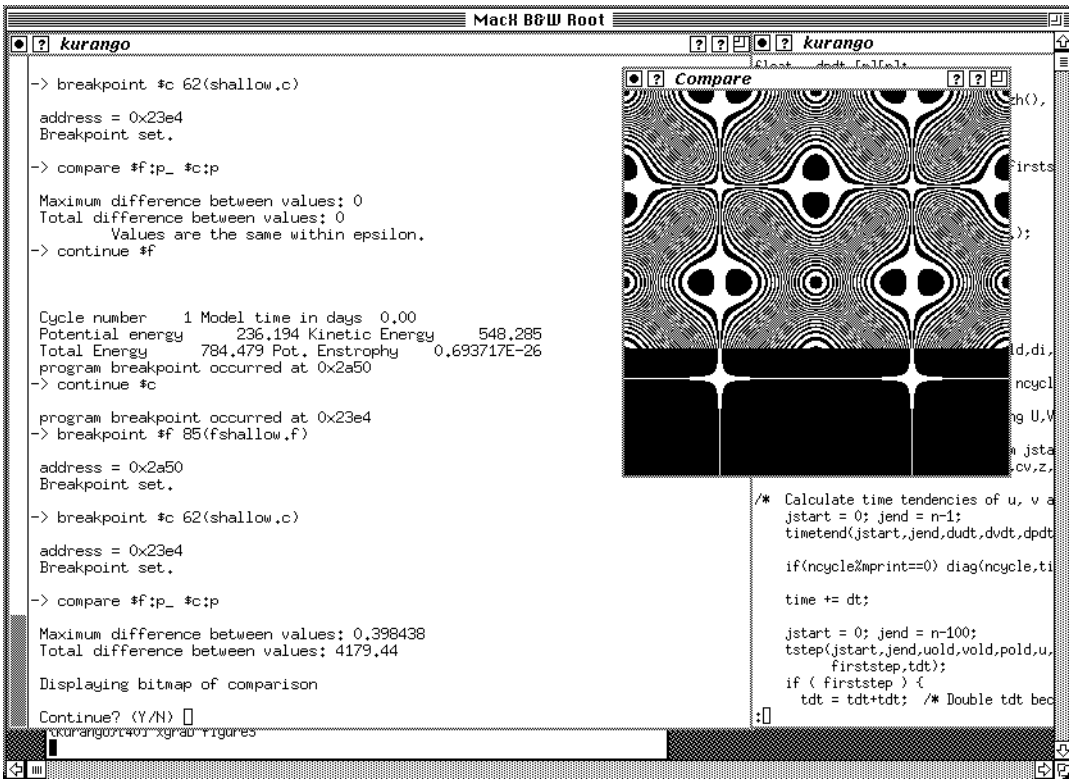

Figure 7: Premature Loop Termination

3.5 Automatic Generation of Assertions

Relative debugging is commonly performed as an iteration of several steps. First, the assertions are specified. Although the user can specify the assertions interactively, a script file is usually used instead. The programs are started to detect any potential differences. After the differences are detected, their source is investigated. As a result, the source code of one of the programs is modified and the script is run again. Because assertions in the script rely on locations in the source file, they must be updated every time the position of these locations changes due to user modifications. In general, the script file with assertions must be updated after every compilation.

Guard can automate this process of generating script files, so that no user intervention is required. The underlying mechanism is a list of containers, which can be generated at compile time. No modifications to the compiler or other system tools are required for list generation. By using a Guard command during debugging, lists from different programs can be combined to form assertions, which can be used exactly as regular, user specified assertions.

Automatic generation of assertions is illustrated in Figure 8. A container is specified in the source code by a pragmatic comment, which contains the name of this container and the variable to be compared. During each compilation of the program, a simple preprocessor traverses the source code and extracts the list of containers, one container per pragmatic comment. Each element in the list contains the container name, its location and the variable name. The container and the variable names are specified in the pragmatic comment by the user. The location is determined by the preprocessor as the line number of the corresponding pragmatic comment in

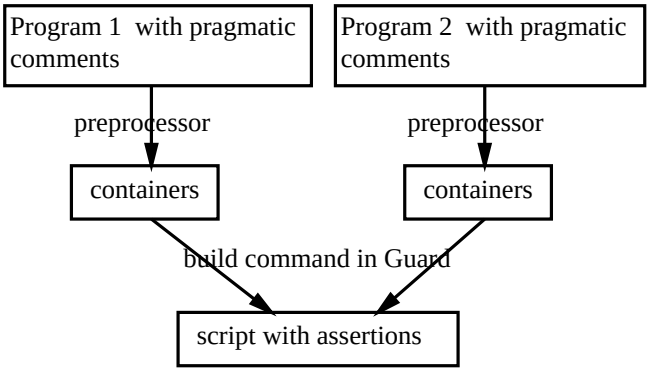

Figure 8: Automatic Generation of Assertions

```
200.    ...
201.        do 300 j=1,n
202.          do 200 i=1,m
203.            uold(i,j) = u(i,j)+alpha*(unew(i,j)-2.*u(i,j)+uold(i,j))
204.            u(i,j) = unew(i,j)
205.    200   continue
206.    300 continue
207.    C$CONTAINER test1 uold
208.    C$CONTAINER test2 u
209.    ...
```

Figure 9: Specification of Containers in Fortran

the source code.

Guard uses the `build` command to combine two container lists:

```
build <process_1>::<file_1> <process_2>::<file_2>
```

The command takes two files with containers, forms assertions, and associates them with processes. Each assertion requires the matching of two containers. This matching is based on container names, specified in pragmatic comments by the user. Containers with the same name, but from two different files, are joined in a single, complete assertion. The naming assures that correct containers are matched regardless of the different source code organization of the two programs. The matching is preserved even when source code is completely reorganized.

The generation of assertions is illustrated on two programs written in Fortran and C. A section of the Fortran program, augmented with pragmatic comments for specifying assertions, is shown in Figure 9. The corresponding section in the C program is shown in Figure 10. Array `uold` in Fortran corresponds to array `vold` in C and array `u` corresponds to array `v`. Although the programs have different structure, their results are expected to be similar. This expectation is specified in pragmatic comments with containers `test1` and `test2`, defined in both programs. From lists of containers, generated automatically during the compilation phase, the `build` command in Guard will generate the following assertions:

```
267.    ...
268.    for (j = jstart; j <= jend; j++){
269.      for (i = 0; i < m; i++){
270.        vold[i][j] = v[i][j]+alpha*(vnew[i][j]-2.*v[i][j]+vold[i][j]);
271.      }
272.    }
273.    /*$CONTAINER test1 vold */
274.
275.    for (j = jstart; j <= jend; j++){
276.      for (i = 0; i < m; i++)  v[i][j] = vnew[i][j];
277.    }
278.    /*$CONTAINER test2 v */
279.    ...
```

Figure 10: Specification of Containers in C

```
assert f::uold@fshallow.f:207 = c::vold@cshallow.c:273
assert f::u@fshallow.f:208 = c::v@cshallow.c:278
```

There is no difference between these assertions and user specified assertions. Both types of assertions can be arbitrarily mixed and verified with a single `verify` command.

Automatic generation of assertions is beneficial in several ways. It simplifies program maintenance by removing tedious manual construction of assertions. The user must maintain only pragmatic comments in the source code, which change infrequently. An additional advantage is that containers can be specified independently for each program. The `build` command in Guard takes into account only containers with a matching pair, other containers are ignored.

3.6 Execution Traces

For debugging purposes, it is sometimes impractical or impossible to execute a real program. The program might require a machine or an environment that is not operational or the program itself might be unavailable. In such cases, the user can use *execution traces*, provided by Guard. With execution traces, the program is run only once and the information about its execution is stored as an execution trace on a disk. The trace can be used later to supply variable values instead of running the original program.

An execution trace can be generated by Guard during any debugging session. Independently for each process being debugged, the user can turn on or off the generation of an execution trace. If the execution trace is turned on, then Guard will output the trace to a file. The file contains all the information, necessary to perform a debugging session for that particular process. The trace consists of containers and the values of their corresponding variables as encountered during the execution. As long as the set of containers and the input values remain the same, the trace can transparently replace the original process at a debugging session.

The generation and the use of traces in Guard is seamlessly integrated with other commands. The tracing capability is added with no change to any of the existing commands. Trace generation is controlled by the `traceon` and `traceoff` commands:

```
traceon <process_name> <file_name>
traceoff <process_name>
```

When the tracing is turned on, the relative debugger automatically saves the trace in tracing file `<file_name>`. The trace contains the values of all the variables, used in any comparison or assertion.

Normally, Guard executes two programs concurrently. However, an execution trace belongs to a single program, so it is desirable to be able to run only that program to produce its execution trace. To provide this capability, Guard supports empty processes. Instead of using the `invoke` command, the second process can be declared *empty*:

```
empty <process_name>
```

Guard ignores all commands for an empty process, except comparisons. If a comparison is performed between an empty and a nonempty process, then the trace will be generated for the nonempty process, if requested. A comparison with an empty process reports no differences, so the trace generation is transparent to the user. Using empty processes, a trace can be generated by running a single process without any changes in assertions.

The trace is used during a debugging session through `ghost` processes. For a ghost process, Guard reads its execution information from a trace file, instead of running a real process. A ghost process is created by a `ghost` command:

```
ghost <process_name> <file_name>
```

For each comparison performed, the container and the value of its variable are read from the trace as if they would be generated by a real process. If the types of structures to be compared do not match or if differences in values are detected, Guard will print an error message and display differences.

The generation and the use of traces does not require any significant changes in the script files. The user specifies a set of assertions and generates the relevant trace to the disk. As long as the set of assertions remains the same, the trace of the reference program can be used in verifying the assertions for the development program. Another trace must be generated only when the set of assertions changes. Because there is no need to execute a real process, execution traces can simplify and speed up relative debugging.

4 Implementation of Guard

4.1 Implementation Issues

Relative debugging requires several advanced capabilities, some of which are not normally available in traditional debuggers. The most important novel capabilities are concurrent control of at least two processes and the capability to operate in a distributed and heterogeneous environments.

Debuggers usually use specialized debugging primitives. The primitives, provided by the operating system, allow debuggers to control other processes and to manipulate their internal state [7, 16]. Guard could use these primitives to carry out debugging operations. This approach would require a fairly straightforward extension to existing debuggers.

However, one of the main powers of relative debugging is the ability to debug programs in a distributed and heterogeneous environments, with programs running on different types of machines. Standard debugging primitives provide very little support for distributed debugging. These primitives are highly system dependent, so it is nontrivial to implement debuggers for heterogeneous environments [21, 22, 26].

To successfully operate in a distributed and heterogeneous environment, Guard uses a different approach. To carry out its debugging operations, it utilizes a system, called Dynascope. Dynascope provides debugging servers, which can be accessed through a system independent debugging interface [31]. These servers implement primitives for building debuggers and other similar applications in distributed and heterogeneous environments. Dynascope is described in the next section.

4.2 Dynascope

Dynascope implements debugging servers, which can be used to build sophisticated debugging and monitoring applications. Such applications are becoming crucial in dealing with increasingly complex software [5, 9, 10, 11, 13, 15, 17, 19, 20].

By hiding system dependencies, Dynascope provides a procedural interface, which is independent of the underlying operating system. Debugging primitives can be executed in distributed and heterogeneous environments. All the necessary communication between processes in a distributed environment is handled internally by Dynascope and is transparent to the user.

Dynascope implements the primitives for: process control; state access; breakpoint handling; tracing; and dynamic loading and linking. Some of the Dynascope primitives are shown in Figure 11. The Dynascope interface is described in detail elsewhere [31].

Although powerful debugging primitives usually require interpreted environments [17, 23, 24, 29, 32, 33], Dynascope operates in traditional, compiled environments. It is compatible with existing compilers, linkers, and other development tools. Dynascope has no special provisions for optimized code. It always retrieves variable values from the program's address space in the main memory. If the right value is kept in a register, but not in the main memory, then an incorrect value will be retrieved.

The structure of Dynascope is shown in Figure 12. Dynascope consists of two components for building debuggers, the *client library* and the *debugging server*. The client library is associated with each debugger, the debugging server with each program being debugged. The communication between the client library and the debugging server is carried out using standard operating system features, such as signals, sockets, and the TCP/IP protocol.

The client library provides a set of procedures, which are called by the debugger to carry out the debugging operations. The library is linked with the debugger in a single executable program. The library handles requests from the debugger by: establishing a connection between the debugger and the debugging server; sending debugging requests to the server; and receiving the results.

The debugging server carries out requests from the client library by controlling and accessing the program. Debugging requests are serviced by accessing the user program and by utilizing standard object formats and debugging information. Two designs of the debugging server have

```
Process Control:                          Breakpoints:

invoke   - start a new process            setbreak   - set a breakpoint
attach   - attach to an existing process  delbreak   - delete a breakpoint
detach   - detach from the process        waitbreak  - wait for a breakpoint
kill     - terminate the process
execute  - execute the process            Tracing:
connect  - stop the process
                                          trace      - trace on and off
                                          getevent   - get a tracing event
State Access:                             ldfilter   - load an event filter
                                          rmfilter   - remove a filter
getmem   - read memory                    initfilter - initialize a filter
putmem   - set memory
getstate - read processor state           Dynamic Loading and Linking:
putstate - set processor state
getsym   - get symbol value               ldobject   - load an object file
time     - get execution time             rmobject   - remove an object file
lineaddr - get line address               link       - link a symbol
```

Figure 11: Dynascope Primitives

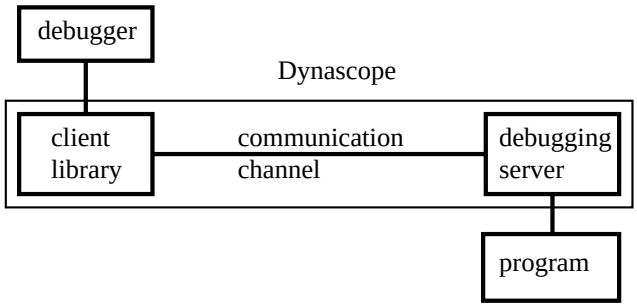

Figure 12: The Structure of Dynascope

Operation	DEC Alpha	IBM SP/2	NeXT NeXTst. Turbo	SGI Challenge	Sun SPARCst. 5
Attach	31.4ms	47.3ms	99ms	53.7ms	57.9ms
Null Command	140μs	100μs	320μs	240μs	255μs
Max. Throughput	36.2Mb/s	40.2Mb/s	4.27Mb/s	18.5Mb/s	13.5Mb/s
Requests/s	4000	6850	2323	3592	3034
Breakpoint	330μs	220μs	620μs	400μs	480μs

Table 1: Dynascope Performance

been implemented. In the first design, the server is a special purpose thread executing in the same address space as the user program. Normally, the user program is executing and the debugging server is inactive. Upon receiving a debugging request, the user program is interrupted and the server is activated. After the request is answered, the user program continues with the execution. The execution of the user program and the server are mutually exclusive. There are no direct procedure calls from the user program to the directing server or vice versa. More details on the implementation of the server are described elsewhere [30]. In the second design, the server runs as a separate program, utilizing system provided debugging primitives for carrying out debugging requests. The second design is simpler to implement, but some of the features, such as tracing or dynamic loading, are not supported, because they are not provided by the system primitives.

Dynascope is implemented on DEC OSF/1, IBM AIX, NeXT Nextstep, Silicon Graphics IRIX, Sun SunOS, Sun Solaris and Linux systems. Implementations on additional architectures are in progress. Debuggers and programs being debugged can be mixed arbitrarily between platforms.

The performance of major Dynascope primitives on various platforms is provided in Table 1. Measurements were taken on machines with minimal load. Each column presents measured performance for one platform. Row *Attach* shows the time required to attach to a process and establish a connection between the debugger and the program. Row *Null Command* gives the execution time for an empty debugging command, demonstrating the minimum command servicing time. Row *Max. Throughput* illustrates the maximum number of bytes that can be transferred between the debugger and the program in a second. Row *Requests/s* gives the maximum number of copying requests that are handled by the debugging server per second. Row *Breakpoint* shows the maximum number of breakpoints reported per second.

4.3 Experiment in Heterogeneous Debugging

As mentioned before, Dynascope provides uniform debugging primitives regardless of the underlying computing platform. Because Guard is built on top of Dynascope, it works on any platform supported by Dynascope. Moreover, it can debug programs that execute on any other Dynascope supported platform.

To demonstrate the heterogeneous nature of Guard, we have performed the following experiment (see Figure 13). In the experiment, Guard successfully compared the execution of programs

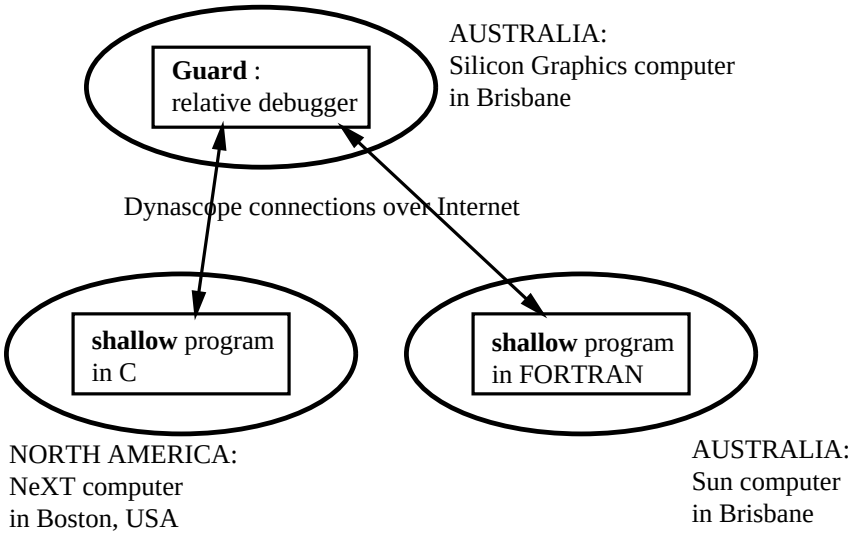

Figure 13: Experiment in Heterogeneous Debugging

across Internet, involving three different types of computers and two different languages. Guard itself was running on a Silicon Graphics computer at Griffith University in Brisbane, Australia. The C version of the program was running on a NeXT computer at Harvard University in Boston, USA. The Fortran version of the program was running on a Sun computer at Griffith University in Brisbane, Australia.

Both programs solved a set of partial differential equations to perform the calculation of waves in shallow water [3]. To perform the experiment, C and Fortran executables were created on the NeXT in Boston and the Sun in Brisbane. These executables were formed by linking user object files with the Dynascope debugging server. The user access rights were set on the NeXT and the Sun, so that Guard could perform a remote login to these machines. The script that was executed by Guard to carry out the experiment is essentially the same as the script shown in Figure 5 with results similar to those in Figure 7.

No modifications of the standard version of Guard and Dynascope were necessary in order to perform the experiment. Dynascope hides all machine dependencies and handles the communication details. Guard code is machine independent and only recompilation is required to port Guard from one platform to another. Because Dynascope provides high level debugging primitives, no extensive system level knowledge was required for implementing Guard.

5 Conclusions

Guard is being successfully used in software development. Several in-depth case studies of using Guard on practical problems are described elsewhere [1, 2]. In those studies, Guard reduced the time, necessary to locate errors in programs with several ten thousand lines of code, to less than an hour. In one case, it found discrepancy between two programs, when no difference was expected. It is estimated that traditional methods of program debugging and testing would

often require weeks to perform similar tasks. Our initial experience with Guard is thus extremely encouraging. We expect that relative debugging will significantly reduce the time required for the evolutionary software development.

At the implementation level, several extensions to Guard are currently being planned. We are working on support for providing complex control of processes. The support will be based on the dataflow approach to debugging [25]. This approach will also increase the power of assertions. Assertions will be able to dynamically determine which containers to match, depending on the run time values from the programs. Additional topics for future research involve issues related to interactive input and output and the coverage of test data. So far, our work has concentrated on numerically intensive applications. We plan to extend relative debugging to distributed, interactive business applications, which require a high degree of reliability. Examples of such applications include bank networks and air-traffic control systems. By using techniques from relative debugging, a new version of software could be run in parallel with the existing version for extended periods of time. The execution of the new version would be compared to the existing version, which would be performing the real work.

At the conceptual level, our experiment with Guard in heterogeneous debugging has successfully demonstrated that it is possible to provide a uniform debugging interface. Such an interface makes possible many novel applications in heterogeneous computer networks. For example, to support the porting of applications to new computing platforms, verification servers could be set up, which would provide known behavior for testing the development programs. By providing the capability of comparing programs in different languages, relative debugging might make practical advanced techniques, such as algorithmic debugging [28]. These techniques could utilize executable program specifications. The reference program can be written in a high level declarative language specifying the program behavior. Although such reference programs would execute much slower than the development version, they could provide executable specifications for the development version. By comparing the execution of the high level specifications and the development version, relative debugging could be used to verify the conformation of the development version with its specifications.

Relative debugging utilizes the fact that most software is being developed through relatively small, evolutionary changes to the existing code. By providing support for this common software development activity, relative debugging has the potential to significantly reduce time consuming phases of software debugging and testing.

Acknowledgments

This work has been supported in part by the Australian Research Council. Lisa Bell has performed much of the programming necessary to implement Guard and proposed the term relative debugging. Andrej Šali arranged computer access in USA. Ian Foster, John Michalakes and Larry Snyder provided many useful suggestions for extensions to Guard. Comments from anonymous reviewers were very helpful in improving the paper.

The information on the availability of Guard can be obtained on WWW:

`http://www.cit.gu.edu.au/~davida/guard.html`

References

[1] D. Abramson, I. Foster, J. Michalakes, and R. Sosič. Relative debugging and its application to the development of large numerical models. In *Supercomputing'95, San Diego*, December 1995.

[2] D. Abramson and R. Sosič. A debugging tool for software evolution. In *CASE-95, Toronto*, pages 206–214, July 1995.

[3] 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.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[5] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in Parasight. In *Proceeedings of the ACM/SIGPLAN PPEALS 1988*, pages 21–30. ACM, 1988.

[6] G. Ashby, L. Salmonson, and R. Heilman. Design of an interactive debugger for FORTRAN: MANTIS. *Software–Practice and Experience*, 3(1):65–74, January-March 1973.

[7] M. J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[8] R. M. Balzer. EXDAMS - EXtendable Debugging and Monitoring System. In *AFIPS Conference Proceeedings, Vol. 34*, pages 567–580. SJCC, 1969.

[9] T. E. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[10] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. *OOPSLA '93 Proceedings, Sigplan Notices*, 28(10):65–82, 1993.

[11] I. J. P. Elshoff. A distributed debugger for Amoeba. In *Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 1–10. ACM, 1988.

[12] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings Supercomputing-93, Portland, Oregon*, pages 462–471. IEEE, 1993.

[13] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report 1103, Dept. of Computer Science, University of Wisconsin, Madison, 1993.

[14] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 1990.

[15] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[16] T. J. Killian. Processes as files. In *Proceedings of the Summer 1984 USENIX Conference*, pages 203–207. USENIX, 1984.

[17] A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 338–352. ACM, 1991.

[18] L. Lopriore. A user interface specification for a program debugging and measuring environment. *Software–Practice and Experience*, 19(5):437–460, May 1989.

[19] J. E. Lumpp Jr., T. L. Casavant, H. J. Siegel, and D. C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483. IEEE, 1990.

[20] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman. Tools for distributed application environment. *IEEE Computer*, 24(8):42–51, August 1991.

[21] J. May and F. Berman. Panorama: A portable, extensible debugger. *ACM/ONR Workshop on Parallel and Distributed Debugging, Sigplan Notices*, 28(12):96–106, 1993.

[22] P. Maybee. NeD: The network extensible debugger. In *Proceedings of the Summer 1992 USENIX Technical Conference*, San Antonio, 1992.

[23] T. G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[24] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[25] 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.

[26] N. Ramsey and D. R. Hanson. A retargetable debugger. In *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 22–31. ACM, 1992.

[27] E. Satterthwaite. Debugging tools for high level languages. *Software–Practice and Experience*, 2(3):197–217, July-September 1972.

[28] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[29] T. Shimomura and S. Isoda. Linked-list visualization for debugging. *IEEE Software*, 8(3):44–51, May 1991.

[30] R. Sosič. Design and implementation of Dynascope, a directing platform for compiled programs. *Computing Systems*, 8(2):107–134, Spring 1995.

[31] R. Sosič. A procedural interface for program directing. *Software–Practice and Experience*, 25(7):767–787, July 1995.

[32] W. Teitelman and L. Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–33, April 1981.

[33] A. P. Tolmach and A. W. Appel. Debugging standard ML without reverse engineering. In *Proc. ACM Lisp and Functional Programming Conference '90*. ACM, 1990.

[34] P. Winterbottom. ACID: a debugger built from a language. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 211–222, San Francisco, 1994.

[35] P. T. Zellweger. Interactive source-level debugging of optimized programs. Technical Report CSL-84-5, Xerox PARC, 1984.