(appeared in Computing Systems, Vol. 8, 2, pp.107-134, MIT Press, Spring 1995.)

The Dynascope Directing Server: Design and Implementation [1]

Rok Sosič
School of Computing and Information Technology
Griffith University
Nathan, Queensland 4111, Australia
E-mail: sosic@cit.gu.edu.au

Abstract

As computer systems are becoming increasingly complex, directing tools are gaining in importance. Directing denotes two classes of activities, *monitoring* and *controlling*. Monitoring is used for collecting information about the program behavior. Controlling is used to modify the program state in order to change program's future behavior. Some characteristic directing tools are debuggers and performance monitors.

Dynascope is a directing platform, which provides basic monitoring and controlling primitives. These primitives are used in building advanced directing applications for networked and heterogeneous environments. Dynascope is integrated with existing programming tools and uses only generic operating system and networking primitives. This paper describes the design and implementation of the directing server, the central component of Dynascope. Dynascope is being used in several applications, including relative debugging, steering agents, and simulator testing.

1 Introduction

Increasingly complex software requires sophisticated directing tools [2, 5, 8, 11, 16, 18, 23, 26, 28]. These tools perform activities, which can be classified as *monitoring* and *controlling*. Monitoring is used for sampling of the program state or tracing of the program execution. During monitoring, information about the program behavior is collected without changing the program's semantics. Controlling is any activity that changes the program and its future behavior by modifying the program's state, including its variables and code. An important application of controlling can be found in *program steering*, which means controlling the execution of long-running, resource-intensive programs [14]. Because monitoring and controlling activities are closely related, we can combine them under a single term, *directing* [40]. Two types of programs are involved in directing: programs that perform directing and programs being directed. The former are called *directors* and the later are *executors*. Directors are constructed specifically to perform a particular directing task. Examples of directors are debuggers and performance monitors. Any user program can serve as an executor.

Directors are complex programs, because they depend on all major components of computer systems: computer architectures, operating systems, networking, and existing programming tools. Their complexity is compounded by the heterogeneous nature of most computing environments. Significant programming efforts are spent on reimplementing similar tools on different systems. A system independent platform for building directing tools would simplify the tool construction. Programming resources could be redirected toward building better tools, capable of operating in heterogeneous environments.

Advanced directing tools are often implemented in environments that provide interpreted execution [21, 23, 32, 33, 34, 39, 47, 48]. Tools in these environments utilize the underlying interpreter

[1] This research was supported in part by the Slovene Ministry of Science and Technology and the University Research Initiative Scheme at Griffith University.

which is extended with specialized support for debugging or performance monitoring. Such an interpreter does not exist for compiled programs, which are directly executed by hardware, so a different approach is required.

Directing applications for compiled programs are normally built by using dedicated primitives in the operating system, such as `ptrace`, `profil`, and `/proc` [3, 13, 22]. These primitives are intended for building traditional directing applications: debuggers and profilers. They provide execution control and access to the process state. Their main limitations are different primitives on different computing systems, a low level and restricted range of primitives, and no support for networks. Their functionality is often insufficient for providing advanced directing applications in heterogeneous environments.

Dynascope provides directing primitives for compiled programs. The primitives operate in networked and heterogeneous environments. Dynascope uses only generic operating system primitives for its implementation. It is compatible with existing development tools and simple to use. It is being successfully applied in several applications by programmers with minimal system specific knowledge. Using Dynascope, these programmers built innovative directing tools for heterogeneous environments. Some projects include: dynamic tracing and visualization of program execution; relative debugging which compares execution of multiple programs in different languages, running on different types of computers; and steering agents for scientific simulations. This paper concentrates on the design and implementation of the directing server, which is the central component of Dynascope.

Section 2 gives a short overview of directing primitives. Section 3 describes the design and implementation of Dynascope. Section 4 describes some applications of Dynascope: relative debugging, steering agents, and simulator testing. Performance measurements are presented in Section 5. Section 6 provides a discussion of the current status of Dynascope. Related work is discussed in Section 7. Section 8 concludes the paper.

2 Dynascope Primitives

Dynascope provides a set of directing primitives. These primitives are used by directors to perform monitoring and controlling operations. The primitives operate across networks. This section provides a summary of Dynascope primitives. A detailed description and the use of the primitives is given elsewhere [41].

Most elementary are primitives for *execution control*. These primitives are used by the director to establish a connection with the executor or to get its attention. Using these primitives, the director can attach to a running program, start a new program, and stop or resume program's execution.

Other primitives are classified into several groups: state access, breakpoints, tracing, and dynamic loading and linking. These primitives require that the executor is waiting for directing commands. This requirement guarantees that the user program is not executing and changing its state, while it is being examined or modified by the director.

Directing primitives for *state access* manipulate the executor's state. The director can obtain or set values in the executor's address space, including data and processor registers. Additional primitives provide addresses of source lines and addresses of global symbols to assist in calculating breakpoint locations and variable addresses. Using *breakpoint* primitives, the director can set up and delete breakpoints or catch breakpoint execution. *Tracing* primitives access and manage tracing events. Tracing complements breakpoints by providing more powerful monitoring constructs at a cost in the execution time. Primitives for *dynamic loading and linking* provide facilities for dynamically changing program code. The director can load new object files in the executor's address space. Symbols from these object files can be linked with the program code.

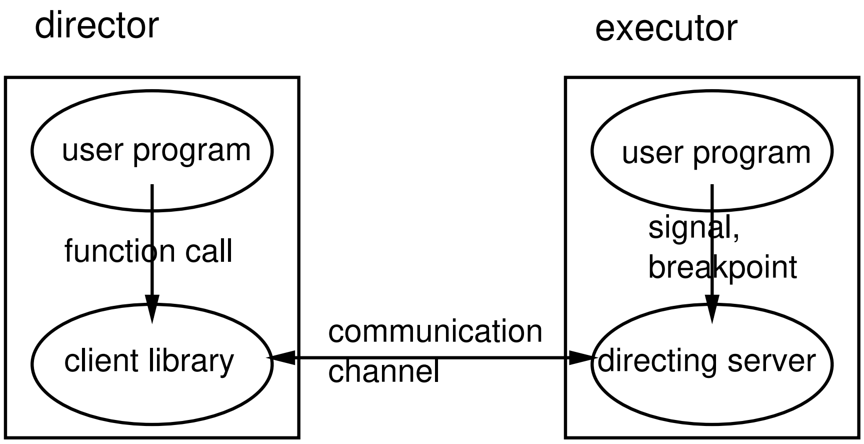

Figure 1: Overview of Dynascope

3 Design and Implementation

3.1 Overview

An overview of Dynascope is shown in Figure 1. Dynascope is implemented in two components, a *client library* and a *directing server*. The client library provides a set of procedures, which are called by directors to perform Dynascope primitives. Procedures in the library send requests to the executor and return the results. Each executor has its own directing server which accepts and carries out directing requests for that particular executor.

The implementation of the client library is straightforward. Procedures in the library pack their arguments, implement the communication protocol with the directing server, and unpack the results.

The directing server contains most of the Dynascope complexity. It isolates system dependent features and provides a system independent interface to the client library. The system independent interface enables the portability of directors and their operation in heterogeneous environments. Only recompilation is required to port a director from one platform to another. Directors can direct simultaneously several executors, each executing on a different machine. As a result, directors and executors can be arbitrarily mixed on any Dynascope supported computing platform.

In order to carry out directing requests, the server must be able to perform several operations. These operations include: halting and resuming the execution of the user program; accessing and modifying the address space of the user program; and handling of breakpoints and tracing primitives. Two approaches can be used to implement the directing server. One approach uses operating system primitives, such as `ptrace` and `/proc` [3, 22]. The second approach uses only generic operating system primitives and avoids the use of special debugging primitives. Each approach has its advantages and disadvantages. The main advantage of the operating system primitives is a relatively simple implementation of the directing server. Disadvantages involve limited functionality. Operating system primitives might not support functionality that is required by the directing server or the primitives might be too limited in their scope. For example, in some operating systems primitives can be applied only to children processes or they can access only one word in the user program at the time. Both limitations would greatly restrict the applicability of directing servers. If operating system primitives are replaced by a user level implementation, disadvantages and advantages are exchanged. Although the initial implementation of the directing server is more complex, it is easier to incorporate extensions. An additional advantage of the second approach is that the directing server can be placed directly in the address space of the user program. This arrangement eliminates expensive process switches between the directing server and the user program, so the servicing of directing commands can be faster.

Dynascope uses the second approach. Each executor is a single process, consisting of a user program to be directed and its directing server. The directing server is implemented as a distinguished thread running in the executor's address space. Because consistency problems can arise otherwise, the server and the user program never execute concurrently. As a result, if no directing is being performed, the server does not impose any overhead, except for occupying the address space. The directing server is activated by directing requests. When a directing request is received, the user program is interrupted and the server thread is started. The interruption is transparent to the user program, because there are no direct calls from the user program to the directing server. The server continues servicing directing requests, until a continuation of the user program is requested. At that point, the server thread returns control to the user program which continues executing.

The directing server is implemented entirely at the user level. It requires only generic operating system primitives, which are used for interprocess communication, remote execution, and signal handling. To build directors and executors, user programs are linked with the client library or the directing server, respectively.

Main issues in implementing Dynascope are the activation of the directing server, the managing of a directing session, and the implementation of directing primitives. These issues are discussed in the following sections.

3.2 Activating the Directing Server

The directing server is implemented as a distinguished thread in the executor, sharing the address space and control with the user program. The server can be activated externally or internally. An external activation is done by a director through issuing a directing request. An internal activation is performed when the user program executes a breakpoint. Each server activation performs a context switch from the user program to the directing server. The server exits by restoring the context of the user program.

The activation and exit from the directing server are performed through signal handlers [10]. An asynchronous signal from the director activates the *entry signal handler*. The signal handler saves a partial state of the user program and redirects the execution to the *server prolog*. The server prolog completes the saving of the user state and replaces the user stack with the server stack. A separate server stack simplifies tracing of the user program. The server prolog passes the execution to the *server entry point*.

The server entry point is the server's main loop. If requested, a communication channel is established with the director before the loop is entered. The server's loop reads directing requests and services them. When the director requests the continuation of the user program, the server performs a context switch back to the user program. The server calls the *server epilog*, which restores the user state and generates the exit signal. The *exit signal handler* returns control to the user program, which continues without being aware of the interruption.

Signal handlers, which perform context switches between the user program and the directing server, allow a relatively portable and simple implementation. In older versions of Dynascope, the directing server used only user level code to exit from the directing server. These user level implementations were complex, highly system dependent, and unreliable. The exit from the server is a critical section, because another directing request might be issued before the exit is completed. In existing versions of Dynascope, the server always exits through a signal handler, which guarantees atomic execution of the exit code, free from signal interruptions. This implementation has proved to be simple and reliable. It has a system independent high level logic, which improves its portability.

The directing server must be initialized, before the user program starts executing. The initialization can be done by a special start-up code which replaces the system supplied code. The special code can be inserted in the executor during the linking of the user program or by directly modifying the executable. The start-up code sets up relevant signal handlers, interprets the command line, and initializes server's variables. Because the command line is shared between the user program and the directing server, a dedicated marker is used to distinguish server's arguments from user's arguments. The marker is specified by a sequence of characters which is unlikely to occur in the user command line. After being processed by the server, the server's arguments are removed from the command line, so that they are invisible to the user program. The most important server's argument activates the directing server before the user program is started. This argument enables the director to manipulate the user program before it starts executing. Other server arguments are used mostly for testing purposes.

3.3 Directing Session

A directing session is established when a director attaches to an executor. The session is active until the director detaches itself. At most one director can be attached to an executor at the same time, but each director can be attached to several executors. The restriction to one director at the time prevents the interference between several directors, modifying a single executor.

During a directing session, Dynascope maintains a communication connection between the director and the executor. The connection transmits directing requests to the executor and returns results to the director. Although the director is logically a client of the directing server, it acts as the connection server. In a previous implementation, the executor was the connection server. This arrangement required a complex protocol in the directing server for dealing with faulty directors and for resolving the race problems among directors, trying to connect to the same executor. The existing solution simplifies the directing server which makes it more reliable.

To establish a directing session, the director must interrupt a running program and open a connection to the program. The session is established as follows (see Figure 2). The director opens a communication channel and sends the channel address and a request for attach to the executor. The channel address is communicated through a file, identified with each executor. The file is created only when necessary. Establishing a directing session through a file solves the problem of races between several directors requesting a connection to a single executor. Because writing to a file is atomic, the information in the executor's file is always consistent. When several directors request a connection at the same time, one of them succeeds by receiving a response from the directing server, while the others receive a timeout after the executor fails to respond.

After the executor's file is created, a request for attach is sent as a signal which activates the directing server. Upon catching the signal, the directing server reads the executor's file and connects to the specified channel. Communication channels use sockets with TCP/IP. Directing commands and their results are transmitted across sockets by an ftp-like protocol [46]. If the director and the executor run on two different machines, then the director uses remote file copy and remote shell execution to access the executor's file and to send signals. In future implementations, the remote execution facilities might be replaced by a specialized server. The remote execution server could provide an increased level of security for handling the setup of directing sessions and transmission of signals.

A separate signal is being used during a directing session to stop an asynchronously running executor. Two separate signals, one to establish a session and one to interrupt the executor during the session, enhance the robustness of the protocol. Previous implementations of Dynascope used only one signal for both purposes, which proved to be extremely fragile. Errors in directors could easily break executors. For example, if the director unexpectedly terminated, it could leave the executor in an unrecoverable state. Because the existing implementations use a dedicated signal to

```
director                                  executor

<executing>                               <executing>
- open a communication channel
- create the executor's file
- write channel information to the file
- send a signal
- wait for a connect
                                          - catch the signal
                                          - activate the directing server
                                          - read channel information
                                          - delete the file
                                          - connect
- send a command
- wait for a response
                                          - receive the command
                                          - send a response
- send a command
- wait for a response
                                          - receive the command
                                          - send a response
...                                       ...
- send an "execute" command
<executing>                               - receive the command
                                          <executing>
```

Figure 2: Establishing a Session

establish a session, another director can attach to the executor and continues with the directing.

The directing server can receive a request for a new session while another session already exists. In this case, the existing session is terminated and a new session is established instead. This policy allows directors to attach to executors which are in an undefined state, possibly because of errors in a previous director or in the executor itself. The policy provides more flexibility than a fixed method for conflict resolution in the server. Because directors are responsible for resolving potential access conflicts, they can implement a conflict resolution method which is the most appropriate for their application. So far in practice, the directors have been mostly initiated by human users and the problem of conflicts has been insignificant.

3.4 State Access

The implementation of commands for state access is straightforward. The director provides an address in the executor's address space, the address of a buffer in the director's address space, and the length of the buffer. Based on the command, the buffer is copied from the director to the executor or from the executor to the director. Specialized commands are provided to simplify the copying of dynamic data structures and strings. These commands automatically allocate the necessary space and, in the case of strings, determine the buffer length.

Besides accessing the address space of the executor, the director can access or modify the processor registers. Because these registers are always captured and restored on entering or exiting the directing server, it is trivial to provide these registers for manipulation by the director.

Two directing primitives assist in mapping source code to machine addresses. The first primitive returns the address of a global symbol. The second primitive returns the address of a source line. These two primitives are used in setting breakpoints and for accessing global symbols. They are implemented by searching symbol and debugging tables which are loaded in the directing server.

3.5 Breakpoints

Breakpoints can be set by a director at any instruction in the user program. When the user program executes a breakpoint, the program is suspended, the breakpoint is reported to the director, and the directing server is activated.

Two different approaches have been tried for implementing breakpoints. The first approach uses only user level code, while the second approach uses a system provided breakpoint trap.

In the first approach, breakpoints are implemented using the *breakpoint code* and the *breakpoint handler* [16, 20]. To set a breakpoint, instructions at the breakpoint address are replaced with the breakpoint code which redirects control to the breakpoint handler. The breakpoint handler saves the user state and activates the directing server. The main advantage of this approach is the speed of breakpoint handling, because no system calls are involved. However, the approach has several disadvantages. The breakpoint code, which is implanted in the user code, can be longer than one instruction. On Sun SPARC, for example, three instructions are required for the breakpoint code. The breakpoint code thus covers instructions other than the breakpoint instruction. If the execution of any of these instructions is attempted, while the breakpoint is set, the resulting behavior of the user program is unpredictable. A similar problem can occur, if a directing request is received, while the breakpoint code is executing.

Later versions of Dynascope use a special, system provided breakpoint instruction for the breakpoint code. These breakpoints occupy only one instruction slot. When the breakpoint instruction is executed, it causes a system trap, which calls the trap handler. From the trap handler, the activation of the directing server is identical to an external server activation. The trap handler passes control to the server prolog which jumps to the server entry point. The implementation of breakpoints through the trap handler is simpler than the first approach, which uses only user level code to handle breakpoints. Because breakpoints activate the server in the same way as directors, the code for the external and internal server activation can be unified. The unified code is significantly more portable and reliable than the older, user level implementation. A breakpoint trap takes only a small amount of the total time required to service a breakpoint. Most of the time is spent in communicating the breakpoint to the director. Therefore, slower execution of the breakpoint handler through a system trap has only minimal influence on the overall breakpoint performance.

Breakpoints require special handling, when the control is returned to the user program. A breakpoint replaces an instruction. This instruction must be executed before the user program continues executing after the breakpoint. The server accomplishes this by replacing the breakpoint with the original instruction. The instruction is executed under the control of the server and the breakpoint is reinstalled. The capability to execute a single instruction is usually provided by the operating system. Unfortunately, this capability cannot be used by the directing server, because it shares the control with the user program. If the directing server implements tracing, which is described in the next section, then the instruction is simply traced. If tracing is not implemented, then single stepping is simulated with breakpoints. Possible addresses are calculated for the next instruction to be executed. Internal breakpoints are implanted at these locations and the user program is resumed. One of the internal breakpoints will be executed, which returns control to the directing server. The server removes the internal breakpoints and switches to the user program.

3.6 Tracing

Tracing allows directors to collect detailed information about the execution of user programs. If a user program is placed in the tracing mode, then each executed instruction produces one or more events. Events are generated by the directing server. The server controls the execution of the user program, produces events, and sends them to the director.

One event is generated for each location modified by the user program. These locations include registers and locations in memory. An event consists of several fields: an event descriptor, the program counter, the instruction register, the changed location's address, the modified value at the address, and the original value at the address before the instruction execution. Events provide extensive information about the program behavior. They can be used to obtain control flow, memory traces, or to perform reverse execution. Although events have machine specific content, their structure is machine independent. All computing platforms provide events with the same fields. Unless the director uses machine specific information, such as operation codes in machine instructions or machine specific registers, events can be processed in a machine independent manner.

Two approaches are commonly used to generate tracing events: single stepping and simulation. *Single stepping* uses hardware support to execute one instruction at the time. The main advantage of single stepping is simple implementation, if sufficient support is provided in hardware and the operating system. There are several disadvantages in using single stepping in the directing server. Normally, the system interface does not enable the server to single step through the user program, because it is part of the same process. Single stepping involves a system call for each executed instruction which causes significant overhead in the execution time. An important disadvantage is that it is expensive to determine which locations change during execution. Partial instruction decoding is required to obtain the addresses of modified locations, which defeats the purpose of single stepping.

Using *simulation*, the hardware processor is emulated in software [9]. This approach eliminates disadvantages of single stepping, although at the expense of increased complexity of implementation.

Dynascope uses *surrogate execution*, a combination of simulation and direct execution. Surrogate execution exploits the fact, that the processor running the simulation and the processor being simulated are the same. Each instruction is decoded to find out its opcode, its operands, and

```
<extract indexes a and b from the instruction>
<save the value of reg[b]>
move r6,r6save        ; save processor register 6
move r7,r7save        ; save processor register 7
move reg[a],r6        ; load user register ra to r6
move reg[b],r7        ; load user register ra to r7
add  r6,r7            ; surrogate instruction: add r6 to r7
move r7,reg[b]        ; save result to user register rb
move r6save,r6        ; restore processor register r6
move r7save,r7        ; restore processor register r7
```

Figure 3: Surrogate Execution of `add` Instruction

the locations modified by its execution. After the decoding, the simulator chooses the relevant simulation sequence. If the instruction is simple, it is simulated. If the instruction is complex, possibly involving condition codes or floating point computations, it is executed by a surrogate instruction. The surrogate instruction performs the same operation as the original instruction, but with different operands. Because of the instruction decoding, the simulator knows the locations of input operands and output results. The input operands are supplied to the surrogate instruction and the surrogate instruction is executed. Its output results are stored to locations that would be modified by the original instruction.

Surrogate execution is demonstrated on a hypothetic instruction that adds register `ra` to register `rb`:

```
add  ra,rb       ; rb = ra + rb
```

`ra` and `rb` can be any of the general purpose registers in the processor. While surrogate execution is being performed, the directing server is active and the user program is suspended. The state of the user program, which consists of an array of register values, is saved in a data structure `reg` in the directing server. Let `reg[x]` denote the saved value of register `rx` from the user program. Simplified code for surrogate execution of the `add` instruction is shown in Figure 3. At the beginning, operand information, in this case indexes of user registers, is extracted from the instruction. The value of user register `reg[b]`, which will be modified by the result, is saved. This original value is later reported in an event. Processor registers `r6` and `r7`, which are used during surrogate execution, are saved to a temporary area. User registers `reg[a]` and `reg[b]` are loaded in processor registers `r6` and `r7`. The surrogate instruction, `"add r6,r7"`, is executed at this point. Although any two registers can be specified in the original instruction, the surrogate execution will always add registers `r6` and `r7`. At the end, the result is saved in user register `reg[b]` and the registers `r6` and `r7` are restored to their original values.

Surrogate execution significantly simplifies the implementation of tracing. It entirely eliminates complex calculations of condition codes and floating point operations in software. An additional advantage of surrogate execution is that it does not involve self-modifying code and subsequent expensive operations to maintain the consistency of the instruction cache. Similar approaches can be found in fast instruction-set simulators [9].

System calls and shared libraries are executed directly. While system calls execute automatically through a trap execution, shared libraries are detected by the directing server. When a shared library is entered, a breakpoint is set at the return address and the library routine is executed directly.

Tracing produces a large number of events, at least one for every executed instruction. The transmission of every event to the director introduces approximately four orders of magnitude of overhead [40]. Many applications require only a subset of events. For example, if the director produces a runtime animation of the function call graph, it needs only events generated by function calls and returns. To reduce the overhead of tracing, the directing server supports filters. Filters are functions that can be dynamically installed in the directing server by the director. After an event is generated, the directing server calls the filter to evaluate the event. If the result is nonzero, the event is sent to the director, otherwise it is discarded.

Commands are provided that load, set and remove filters. An example of the filter that selects only events associated with control instructions is shown below:

```
/* return 1 for control instructions, return 0 otherwise */
eventfilter()
{
  return(evstate[ESAD] == PCADDRESS);
}
```

Because control instructions change the program counter, these instructions generate an event with the address field equal to the address of the program counter. The filter simply compares the address changed, `evstate[ESAD]`, to the address of the program counter, `PCADDRESS`. If the two addresses match, this denotes a control instruction.

Because the communication of events is significantly more expensive than their generation, event buffering considerably reduces the overhead of tracing. With buffering, a few hundred events can be collected together and sent in one block with almost the same overhead as one event. A previous version of Dynascope provided special directing primitives for event buffering. However, a simpler, but functionally equivalent solution can be provided by using a filter for event buffering. The filter collects events and reports a single event when the buffer is full. After the director receives a buffer full event, it copies the buffer from the user program and resets the buffer in the filter. The implementation of buffering with a filter is more flexible than the original solution with a direct implementation in the directing server. Because filters can be changed dynamically, a filter implementation is not fixed and can be easily adapted to suit specific application needs.

The installation of filters is implemented using dynamic loading and linking facilities, described in the next section.

3.7 Dynamic Loading and Linking

At any time during runtime, an external object file can be dynamically loaded and linked with the user program. These modifications to the user program remain permanent, unless they are explicitly changed or reversed. The directing server provides separate constructs for loading and linking.

The loading of an object file takes several steps: sufficient space is allocated from the heap of the executor; code and static data are extracted from the object file and copied to the allocated space; and references to external symbols are assigned values from the already running code. If the object file defines symbols that are already defined in the running code, then the values from the running code are given priority. Therefore, the loading process modifies the object file, but it does not change the previously existing code. For each loaded object file, the directing server builds an entry in the table of object files. The entry consists of the file name and type information, its corresponding static data, its symbol table, and the relocation information. Besides the information about object files, the directing server maintains a separate table of global symbols. For each symbol, the information is maintained about its name, value, and references.

Linking, which takes a file name and a symbol name, changes symbol values and references. When the link command is issued, the value of the symbol from the specified file is made global. All references to the symbol are modified to reflect the new value. Linking is done by directly modifying symbol references in the user program. At runtime, it does not impose any penalty of indirect function calls, usually present in implementations of dynamic linking [15].

Dynamic loading and linking is used in program testing and debugging. Often, two versions of a single procedure are loaded simultaneously. The first one is a production version, optimized for speed, the other one slower test version with possible additional output. At runtime, the two versions can be interchangeably activated without the need to restart the program from the beginning. This approach is useful for long running programs, which run mostly unattended, but require periodical checks on their behavior.

4 Applications

Dynascope enables many novel directing applications. This section describes relative debugging, steering agents, and testing of tracing capabilities in Dynascope. Relative debugging is a new technique which helps in software development and program porting. Steering agents assist with the execution and integration of complex scientific simulations. Tracing capabilities of Dynascope have been tested by comparing the results of tracing with direct execution. Some additional applications can be found elsewhere [41].

4.1 Relative Debugging

Relative debugging enables a comparison of values from two running programs. Usually, one of the programs is a reference version whose behavior is expected to be correct. The second program is a development version whose behavior must correspond to the reference version. The development version can differ from the reference version in several aspects. It can be written in another language, it can run on another platform, or it can incorporate different algorithms which produce the same results.

Relative debugging helps in program development and maintenance. Changes to programs are often incremental and large parts of the program functionality remain the same. A relative debugger can verify that the behavior of the modified program is the same as the original program. The debugger executes both programs concurrently and checks the corresponding data structures. Another important use of relative debugging can be found in porting programs from one platform to another. The behavior of the program on the new platform can be checked for correspondence to the program on the original platform.

We are developing a relative debugger, called Guard [42]. In addition to regular debugging commands, Guard provides commands for relative debugging. Guard can control two processes at the same time. Using Guard commands, the user can specify the corresponding data structures in both processes and locations at which the data structures should be compared. Guard places the breakpoints and runs both programs. After breakpoints are hit, it compares the specified data structures and reports any differences found. Guard provides several facilities to automate this process of program comparison. These facilities involve declarative commands for relative debugging and automatic generation of debugging scripts from pragmatic comments in the source code. Guard can also compare data structures from programs written in different languages. Comparisons automatically take into account different representations, such as a row-major or column-major representation of arrays.

Guard uses Dynascope to carry out debugging tasks. It is a nontrivial application which utilizes the portability of Dynascope directors and their ability to operate in heterogeneous environments. No changes in its source code were necessary to port Guard to all platforms, supported by Dynascope. Only recompilation of Guard was required. Guard and programs being debugged can be arbitrarily mixed on Dynascope supported platforms. For a demonstration, we have successfully compared the execution of a program in C with a program in Fortran. Both programs calculated shallow water equations [1]. The program in C run on a Next computer, the program in Fortran run on a Sun computer, and Guard run on a Silicon Graphics computer. All three computers were connected through Internet.

The experience with Guard demonstrates that Dynascope provides debugging primitives which are portable across computing platforms and capable of operating in heterogeneous environments. In addition, the development of Guard required no knowledge about system dependent debugging and communication primitives. Dynascope was thus successful in abstracting low level system dependencies and in providing high level, system independent directing interface.

4.2 Steering Agents

Although Dynascope includes a collection of debugging primitives, its main purpose goes beyond debugging. One of our projects, which is using Dynascope for nondebugging purposes, is developing steering agents for scientific computations.

Steering agents provide runtime interaction with computationally intensive scientific simulations. Usually, these simulations are performed in a batch mode, producing large files with results. However, it is often desirable to be able to interact with the simulation, while the computation is still in progress. Such interaction could be used to predict the running time of the computation, to check the ongoing status of the simulation, to change some parameters at runtime, or to turn on or off the monitoring of important variables.

Dynascope primitives include machine independent execution control, interprocess communication and access to process states. These primitives are used by steering agents to interact with simulations. Normally, a simulation is running undisturbed. Upon a request from the user, the agent interrupts the simulation and places a breakpoint at a location that is known to have a consistent simulation state. Possible breakpoint locations are loop and function entries and exits. After the simulation reaches a breakpoint, the agent performs whatever task is requested by the user. The task can involve the retrieval of partial results of the simulation or modifications of some simulation parameters.

The use of Dynascope in constructing steering agents has several advantages over more traditional approaches, such as message passing libraries. A steering agent can exercise control over the simulation. As a result, the simulation is not required to contain any support for steering agents. If a message passing library is being used instead, the communication between the simulation and the agent must be explicitly programmed in the simulation. Scientific simulations are usually large programs, whose internal structure is often not very well understood by people using them. The integration of explicit communication constructs can require a significant amount of resources. An additional problem with explicit communication constructs is increased maintenance cost. Because the simulation exists in the original version and a version for agents, two versions must be maintained. Using Dynascope, communication constructs are not hardwired in the simulation at compile time. The communication is specified and controlled by the director at runtime through breakpoints and other Dynascope primitives. This approach is much cheaper to implement, because the simulation code does not require any significant modifications. Dynascope also enables more flexibility in interaction with the simulation, because decisions about agent actions can be postponed until runtime.

Currently, we are developing steering agents for some representative scientific simulations. Agents are capable of steering scientific applications on different computing platforms across networks. We are also exploring scripting capabilities which would automate most of the steering actions and make the agents more autonomous.

4.3 Testing of Tracing Capabilities

Dynascope has been used for testing the implementation of tracing in Dynascope itself. A director has been constructed which executes concurrently two copies of a single program, `A` and `B`. Copy `A` is traced, while `B` is directly executed. The two copies exchange information that would be impossible to obtain only by tracing or only by direct execution. The traced copy provides breakpoint locations for the directly executed copy. On the other hand, the state of the directly executed copy is used to verify the results of tracing.

The main loop of the director is as follows:

```
while (executors_A_B_running()) {
  trace(A,1);                   /* trace one instruction of A */
  getstate(A,stateA);           /* get state of A */
  setbreak(B,stateA[PC]);       /* set breakpoint in B */
  execute(B);                   /* run B */
  waitbreak(B);                 /* wait for a breakpoint in B */
  getstate(B,stateB);           /* get state of B */
  if (!equal(stateA,stateB))    /* compare A and B */
    error(stateA);
}
```

The director traces a single instruction from program `A` at the time. The program counter from program `A`, denoted by `stateA[PC]`, provides the address for a breakpoint in program `B`. Program `B` is then executed until it reaches the breakpoint. After the breakpoint, the states of programs `A` and `B`, which include only processor registers, are compared. If a comparison detects differences, then the error in the simulator is reported by displaying the state of program `A`.

This director was significantly more effective in testing the simulator than alternative methods, such as locating errors by hand or extensive printouts of the simulator operation. A major advantage of the director over other methods is its automatic operation. The director can execute long tests without human intervention. The execution speed during testing is around five orders of magnitude slower than direct execution. Four orders of magnitude are spent in tracing and reporting events. An additional order of magnitude is added by the managing of both executors. If the speed is important, more than two orders of magnitude increase in speed can be obtained by buffering multiple instructions together or by tracing several instructions in one step.

5 Performance Measurements

The performance of Dynascope directing primitives has been extensively measured. Measurements were performed on a NeXTstation Turbo, using a 33Mhz M68040 processor and the NeXTStep 2.2 operating system, and a Sun Sparc ELC, using a 33Mhz Sparc processor and the SunOS 4.1.3 operating system. Results indicate performance for two different architectures, NeXT and Sun, and for two different modes of directing, local and remote. Three configurations are shown: *Next local,* the director and the executor are on the same Next computer; *Sun local,* the director and the executor are on the same Sun computer; and *Sun remote,* the director and the executor are on two different Suns, connected by Ethernet. All measurements were performed on lightly loaded machines. Some operations involve almost no communication between the director and the executor, so the results for the Sun remote configuration are omitted.

Measurement results are presented in Table 1. They show elapsed time, required by a director to perform Dynascope primitives. A user can expect this performance in directing applications.

Row *Attach* shows the time required to establish a directing session between a director and an

Primitive	Next local	Sun local	Sun remote
Attach	85ms	197ms	5,150ms
Connect	1.4ms	1.7ms	2,650ms
Null Command	620us	750us	3,220us
Maximum Throughput	2.13Mb/s	3.45Mb/s	0.39Mb/s
Data Requests/s	1360	1070	220
Breakpoint (internal)	11us	454us	—
Breakpoint (complete)	5.1ms	6.3ms	202ms
Tracing/Execution Ratio I	272	344	—
Tracing/Execution Ratio II	241	525	—
Loading an Object File	18ms	21ms	—
Linking a Symbol	1.85ms	2.34ms	—

Table 1: Performance of Dynascope Primitives

executor. *Attach* involves creating the executor's file, sending a signal, and establishing a TCP/IP connection. It can be observed that there is a significant difference, almost an order of magnitude and a half, between local and remote operations. Local sessions take on the order of 100ms, while the remote session requires more than 5s to establish. Additional measurements showed that *Attach* imposes very little overhead on the execution time of the executor. Dynascope can thus support directors which need to attach to the executor 10 times per second in the local mode or 10 times per minute in the remote mode. *Attach* is required at the beginning of a directing session, so it needs to done only once.

A more significant primitive for application performance is *Connect*. It is used to interrupt an executor, after a directing session has been already established. *Connect* involves sending a signal and receiving a response. *Connect* is important, because it determines the maximum rate at which the user program can be sampled by a director. Results show that local directors can sample user programs around 600 times per second, while remote directors can sample only 20 times per minute. The local sampling rate is high enough to support applications requiring real time human interaction, such as an interactive visualization of a running program. The remote sampling rate is too low for real time interactions, because the sampling can be performed only once every few seconds.

After the director connects to an executor, it issues commands. The rate of command issuing is demonstrated by *Null Command*. The command immediately returns a value to the director without performing any other computation. Row *Null Command* gives the response time of the directing server and involves a message exchange over a TCP/IP. It can be observed that over 1200 commands can be issued per second in the local mode and over 300 in the remote mode. These rates are high enough to support real time user applications, such as an exploration of a static program state, in the local and remote mode.

An important performance indicator is the rate of data transfer between the director and the executor. The first measurement, shown in row *Maximum Throughput*, provides the maximum data throughput. This throughput was measured by copying large blocks of data. It is achieved by Dynascope in applications that copy large arrays of continuous data. The second measurement, shown in row *Data Requests/s*, gives the maximum number of data requests per second that can be handled by the directing server. This number was obtained by copying a lot of small blocks. This performance can be expected in applications that copy a large number of small elements, which cannot be copied as a single block. An example is copying of a linked list, where each element must be identified and copied separately. Measurements show that several Mbytes of data can be copied per second in the local mode and several 100Kbytes in the remote mode. More than 1000 elements of a linked list can be copied per second in the local mode and around 200 elements in the remote mode. These two measurements essentially give the throughput and the turnaround time of the underlying implementation of the TCP/IP protocol.

Two measurements were performed for the handling of breakpoints. The first measurement, shown in row *Breakpoint (internal)*, gives the breakpoint servicing time inside the executor. It shows the time required to perform context switches from the user program to the directing server and back to the user program. A large difference can be observed between Next and Sun. The reason is that the breakpoint handler executes in the user mode on the Next, but requires a system trap to save register windows on the Sun. The second measurement, shown in row *Breakpoint (complete)* gives the complete time of the breakpoint handling. It includes a breakpoint cycle of issuing a breakpoint request by the director, resuming the executor, and receiving a breakpoint report. Dynascope breakpoint handling is thus suited for applications with around 160 breakpoints per second in the local mode and 5 breakpoints per second in the remote mode. Results for *Breakpoint (complete)* demonstrate that system traps on Suns have only limited influence on the complete breakpoint cycle. Because system traps for servicing breakpoints have other advantages, as described in the section on breakpoints, later versions of Dynascope use this implementation.

Tracing involves simulation of machine instructions which imposes execution overhead. The speed of tracing was measured on Stanford benchmarks, a suite of 10 small C programs. Because the overhead of tracing is large, it is assumed that cache effects are not significant and that small programs are sufficient for overhead estimations. Row *Tracing/Execution Ratio I* gives the ratio between tracing and direct execution of non-optimized Stanford benchmarks. Row *Tracing/Execution Ratio II* gives the ratio between tracing and direct execution of optimized Stanford benchmarks. Although the same approach of surrogate execution is used to simulate M68040 and Sparc instructions, measurements show significant differences in the execution speed. Tracing of nonoptimized and optimized versions is from 240 to 270 times slower on the Next and from 340 to 520 times slower on the Sun. The increase in the ratio between the non-optimized and the optimized code on the Sun is attributed to the increasing relative frequency of procedure calls and the corresponding explicit handling of register windows in software. Because tracing in Dynascope has not been optimized for speed, it is expected that its speed could be improved significantly.

Row *Loading an Object File* shows the time required to dynamically load a simple object file. Row *Linking a Symbol* shows the time required to dynamically link a symbol. Results show that linking is around 10 times faster than loading. For some directing applications, preloading of object files can thus save significant time. Times to load and link a filter are similar to their corresponding operations with object files.

6 Discussion

Existing implementations represent the third version of Dynascope. The first version was built around a virtual machine code which was interpreted at runtime [40]. It required a specialized compiler, a linker, and an interpreter of machine code.

The second version works with regular, compiled programs, so no special compilers or linkers are needed [41]. The virtual machine code and the interpreter have been eliminated. Although the internal structure of Dynascope has been completely changed during the transition from the first to the second version, the procedural directing interface remained almost the same. The second version has been implemented on Next and Sun computers.

The third version involved a redesign of the directing server. Before, directing servers have been uniform and implemented from scratch on each platform. The existing server is modular and isolates system specific parts, which makes it easier to port.

Dynascope is currently implemented on several computing platforms. The client library, which does not contain any system specific constructs, has been compiled without changes on the following systems: Sun SPARC, Silicon Graphics MIPS4000, IBM RS6000 and Next M68040. The directing server has modular structure. Primitives for execution control, which enable the director to control the executor, are essential. Other groups, which provide state access, breakpoints, tracing, and dynamic loading and linking, are optional and not necessarily provided by every implementation. The directing server has been fully implemented on Sun SPARC and Next M68040. Partial directing servers, which provide only execution control, state access and breakpoints, but no tracing or dynamic loading and linking, exist on Silicon Graphics MIPS and IBM RS6000.

Dynascope has successfully met design objectives. Directing is performed transparently to the user program. The program's source code is not required to contain any explicit constructs to support directing. The directing server does not significantly slow down the execution of programs that are not being directed. By encapsulating system dependent constructs, Dynascope simplifies the development of directing applications. Directors and executors can be developed using existing program development tools, such as compilers and linkers. Directors are easily ported across computing systems. Directors and executors can run on separate computers and communicate over a network. A director is able to direct programs on different types of computers. Directors and executors can be freely mixed regardless of the type of computing systems they run on.

The largest and most complex Dynascope application is Guard, a relative debugger. It consists of almost 6000 lines of C code. Guard demonstrates Dynascope capabilities in a practical application. Only recompilation was required to port Guard to all Dynascope supported platforms, which confirms the portability of Dynascope primitives. Guard can debug programs running on any Dynascope supported platform, which demonstrates the capability of Dynascope to operate in heterogeneous environments. Guard was developed with no knowledge about system specific debugging primitives. By providing system independent primitives, Dynascope makes the development of directing applications accessible to a much wider programming community.

Current development of Dynascope concentrates mainly on two areas. The first area is expanding the support for programming languages. This support will include primitives for handling language dependent information, such as variable types, structure layouts, and similar. The second area involves additional primitives for heterogeneous systems. All systems, currently supported by Dynascope, have similar architectural characteristics. They have 32-bit words, are big-endian, and use the same representation for floating point numbers. The goal is to provide new primitives for supporting systems with different architectural characteristics, while maintaining the portability and heterogeneity of directors.

7 Related Work

Dynascope aims to provide a unified framework for building directing applications. As a result, it incorporates techniques from a wide range of tools and approaches, such as debugging, monitoring, instruction simulators and dynamic linkers. A discussion of related work is presented here.

Debugging primitives, which form the core of Dynascope primitives, are usually provided by the operating system [3, 22, 38]. Debuggers rely on these primitives to carry out their operations [25, 30, 44]. Because operating systems provide different debugging primitives, debuggers must be carefully engineered to support porting to different platforms [37]. Significant amount of research has been carried out recently to provide high level debugging abstractions and languages [12, 35, 36, 49]. Some debuggers provide debugging in networked and heterogeneous environments by using interpreted languages, which control sequential debuggers on each target machine [29, 31]. Dynascope simplifies the construction of debuggers for networked and heterogeneous environments by providing system independent debugging primitives. These primitives encourage the development of high level debugging languages. Using Dynascope primitives, these languages become portable across platforms and capable of debugging in networked and heterogeneous environments. Dynascope primitives also eliminate the need for a separate debugger on the target machine, which enables faster execution of debugging primitives.

Another important application area for Dynascope is monitoring. Most extensive monitoring capabilities are usually provided in interpreted environments [17, 23, 33]. One approach to monitoring is to use operating system primitives [13]. There primitives provide only a limited range of capabilities. A common approach is to instrument the user program with code, which produces required monitoring information. Instrumentation is incorporated in the source code [7, 45], or in the object code or executable [6, 24, 43]. Some applications modify the user program at runtime [16], using techniques that are similar to user level breakpoints [20]. Monitoring applications consist of several phases: program instrumentation, event generation, event transport, and analysis. Each monitoring application provides its own primitives to carry out these phases, although large parts of functionality are similar for different applications. Dynascope provides common primitives for building monitoring applications, such as primitives to control the executor and access its state. Because monitoring applications can use these common primitives and need to implement only application specific primitives, they require much less effort to build. It is expected that Dynascope will be extended with the most commonly used monitoring primitives in the future, so that they can be directly used by applications.

Instruction level simulators represent a special class of monitors. Instead of running an instrumented program, these simulators generate events by simulating instructions. At the cost of longer execution times or larger space requirements, they are capable of providing more execution details than instrumentation monitors. Some of the simulators, such as Shade [9], provide extensive support for instruction analysis or even a debugging interface to the simulated target machine. Surrogate execution in Dynascope is similar to techniques, used in some of the simulators [4, 9, 27]. Dynascope can be used as an instruction level simulator, if its performance overhead of two orders of magnitude can be tolerated by the application. The overhead is a result of event generation, because Dynascope generates extensive information for every instruction. Tracing performance of Dynascope could be improved significantly by incorporating additional advanced techniques from simulators, such as partial translation or selective tracing of instructions [9].

Dynamic linking provides a structured way to modify program code. Using dynamic linking, executing programs can be changed without being restarted from the beginning. This capability is important in controlling long running programs, because it is often prohibitively expensive or time consuming to restart these programs from the beginning. Dynamic linking can be done by programs modifying themselves [15] or by programs modifying other programs [19]. Dynascope implements the second, two process, model. This model provides more flexibility, because decisions about dynamic linking can be postponed until runtime. Another advantage of a two process model is that a corrupted program can be repaired by another program. It is expected that dynamic linking will gain in importance in future applications of Dynascope.

8 Conclusion

Dynascope demonstrates that a powerful and practical directing interface for compiled programs can be implemented using only generic operating system primitives. An implementation of the directing interface at the user level has several advantages. A unified directing interface hides system complexity and provides a base for building portable directing applications, capable of operating in heterogeneous environments. The interface can be easily extended with advanced features or adapted to specific application needs. Experience with several projects that use Dynascope to build directing applications confirms this. Programmers with almost no knowledge of system programming were able to produce novel applications in a short time. By implementing a simple, system independent interface, Dynascope provides foundations for numerous new directing applications.

9 Acknowledgments

Suggestions from David Abramson and Simon Wail improved several aspects of Dynascope. Simon Wail has implemented the directing server on IBM RS/6000. Chengzheng Sun, Lisa Bell and David Keppel provided numerous useful suggestions on earlier versions of the paper. Comments from anonymous referees were extremely helpful in improving the paper.

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

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

[4] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the Winter 1990 USENIX Technical Conference*, May 1990.

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

[6] M. Bishop. Profiling under unix by patching. *Software–Practice and Experience*, 17(10):729–739, October 1987.

[7] M. H. Brown. Exploring algorithms using BALSA-II. *IEEE Computer*, 21(5):14–36, May 1988.

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

[9] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137. ACM, May 1994.

[10] G. V. Cormack. A micro-kernel for concurrency in C. *Software–Practice and Experience*, 18(5):485–491, May 1988.

[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] M. Golan and D. R. Hanson. DUEL - a very high-level debugging language. In *Proceedings of the Winter 1993 USENIX Technical Conference*, San Diego, 1993.

[13] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices*, 17(6):120–126, June 1982.

[14] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *Sigplan Notices*, 29(9):140–148, September 1994.

[15] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software–Practice and Experience*, 21(4):375–390, April 1991.

[16] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *1994 Scalable High-Performance Computing Conf.*, pages 841–850. Knoxville, Tenn., 1994.

[17] C. L. Jeffery and R. E. Griswold. A framework for execution monitoring in Icon. *Software–Practice and Experience*, 24(11):1025–1049, November 1994.

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

[19] J. Kempf and P. B. Kessler. Cross-address space dynamic linking. In *IEEE Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, 1992.

[20] P. B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 78–84. ACM, 1990.

[21] G. Kiczales and D. G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

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

[23] 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.

[24] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software–Practice and Experience*, 24(2):197–218, February 1994.

[25] M. A. Linton. The evolution of DBX. In *Proceedings of the Summer 1990 USENIX Technical Conference*, pages 211–220, Anaheim, 1990.

[26] 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.

[27] P. S. Magnusson. A design for efficient simulation of a multiprocessor. In *Proceedings of the First International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, January 1993.

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

[29] 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.

[30] P. Maybee. pdb: A network oriented symbolic debugger. In *Proceedings of the 1990 Usenix Winter Conference*, January 1990.

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

[32] M. L. Model. Monitoring system behavior in a complex computational environment. Technical Report CSL-79-1, Xerox PARC, 1979.

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

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

[35] R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A GNU improved programmable debugger. In *Proceedings of the Summer 1990 USENIX Technical Conference*, pages 221–231, Anaheim, 1990.

[36] 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.

[37] 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.

[38] D. D. Redell. Experience with Topaz teledebugging. In *Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 35–44. ACM, 1988.

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

[40] R. Sosič. Dynascope: A tool for program directing. In *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 12–21. ACM, 1992.

[41] R. Sosič. A procedural interface for program directing. *Software–Practice and Experience*, to appear, 1995.

[42] R. Sosič and D. Abramson. Guard: A relative debugger. Technical Report CIT-1994-21, School of Computing and Information Technology, Griffith University, Brisbane, 1994.

[43] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[44] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[45] J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[46] W. R. Stevens. *Unix Networking Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

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

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

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