A Survey of Support For Implementing Debuggers

Vern Paxson
CS 262
Prof. Anderson
October 30, 1990

Abstract

The degree to which hardware and operating systems support debugging strongly influences the caliber of service that a debugger can provide. We survey the different forms in which such support is available. We limit our survey to lower-level debugger design issues such as accessing the debugged program's state and controlling its execution. The study concentrates on those types of support that make overall debugger performance efficient and that support debugger features for ferreting out hard-to-find bugs. We conclude with an overview of state-of-the-art debuggers and a proposal for a new debugger design.

Introduction

Debugging is one of the fundamental aspects of software development. Once software has been specified and designed, it enters a well-known edit–compile–test–debug–repeat implementation cycle. During this cycle there is clearly a need for tools that aid debugging: one study of experienced programmers [1] found that 78% had bugs that took longer than one day to find, and 34% had bugs that took more than a month to find.

Breakpoint debuggers are now widely available. These debuggers allow a programmer to suspend a program's execution at a particular point, inspect and possibly modify the program's state, and then continue the program. In their usual form breakpoint debuggers are adequate for a number of debugging tasks, but they provide little support for particularly difficult bugs, such as those whose effects are not apparent until long after their erroneous action occurs.

More powerful debuggers are available which can aid in rapidly locating such bugs. As we shall see, the degree to which hardware and operating

systems support debugging strongly influences the caliber of service that a debugger can provide. This study concentrates on these support issues and how they affect debugger design.

First we consider what kinds of bugs are difficult to find with standard breakpoint debuggers. The characteristics of these bugs will provide a context with which to evaluate the merits of different debugger features. Next we give an overview of the different features that a debugger can offer, some of which we will explore further, and some of which are beyond the scope of this paper. We then discuss the basic mechanisms underlying breakpoint debugging. With this background in place, we first look at hardware support for debugging, followed by a discussion of the most basic issue in debugger design: whether the debugger executes in the same process as the debuggee[1] or in a separate process; and, if in a separate process, what types of services are available for communication between the debugger and the debuggee. Next we look at systems for supporting checkpointing and reverse execution, which combine to offer a powerful but at the moment rarely available form of debugging support. We then present an overview of some state-of-the-art debuggers and the support they require, and conclude with a summary of the themes that have emerged and how some of them might be integrated into a new debugger design.

Hard-to-Find Bugs

Simple breakpoint debuggers excel at finding certain types of bugs. Any bug that immediately causes a program fault is caught in the act. Bugs involving taking incorrect execution paths can also be instantly found by breakpointing at the beginning of the incorrect path[2]. In general, bugs whose effects are almost immediately manifested can be found with simple breakpoint debuggers by setting a breakpoint at the point of manifestation and then mentally looking back along the execution path a few steps.

Where debugging becomes difficult is with bugs whose effects remain masked for a lengthy period of execution. These include bugs where incorrect values are computed but then not consulted for a while, or invalid data values are accessed, or "out-of-bounds" modifications occur due to pointer misuse.

[1] A program being debugged is a *debuggee*.

[2] Providing that the path is only rarely taken. This will be discussed more fully momentarily.

While bugs that occur in obvious places can usually be found quite easily, if the location where the bug occurs is heavily traveled then it can be very difficult to find the bug using a simple breakpoint due to the number of extraneous breakpoints that the programmer must consciously ignore. For example, if after several thousand calls to a routine, the routine corrupts one of its internal data structures (without causing a fatal fault), then simply getting the program to the particular state where the data structure has just been corrupted can be inordinately tedious. The process is also prone to painful mistakes, where, numbed with boredom, the programmer accidentally allows the debuggee to continue past the desired state, and must begin the process afresh. These sorts of bugs are ideal candidates for conditional breakpoints (see below).

Another type of bug that can be difficult to find is one in which some property of a data type is violated. An example is a link list into which a cycle is erroneously introduced. Debuggers typically have a very low-level view of a program's semantics, operating in terms of the values of individual registers and memory locations. To find these types of bugs efficiently requires the ability to dynamically add high-level assertion checks into the debuggee.

One of the tools for finding these sorts of bugs is the *watchpoint*, which is a type of breakpoint that is triggered whenever a particular memory location is read or written. Another tool is the *conditional breakpoint*—a breakpoint that interrupts the debuggee's execution only when a particular condition holds. A simple type of conditional breakpoint is one that interrupts execution after the breakpoint has been reached n times. Such a facility is very useful for finding bugs in heavily-traveled locations; binary search can be used to locate the precise value of n for reproducing the program state at the onset of the bug. When a conditional breakpoint's condition is expressed in low-level terms it is similar in power to a watchpoint. With some debuggers, though, the condition can be expressed at the same semantic level as that in which the program was written, in which case the conditional breakpoint becomes extremely powerful (though difficult to implement efficiently).

Conditional breakpoints have historically often been extremely inefficient, leading perhaps to an underappreciation of their value [2]. In our survey we will pay especially close attention to support for efficient conditional breakpoints.

A particularly difficult-to-find type of bug is a *sporadic* one: one that is not readily reproducible. These bugs can be due, for example, to referencing

uninitialized memory, to interprocess interactions that have an element of indeterminism in them (typically due to scheduling), or to real-time effects where indeterminism is introduced by the behavior of the world external to the computer.

A very nasty form of sporadic bug is a *Heisenbug*[3]. This is the (somewhat whimsical) name given to a bug whose manifestation disappears when a debugger is used to find it. Timing-related bugs and those involving illegal memory references are examples.

The principle tools for finding sporadic bugs are checkpointing and reverse execution. The principle method of avoiding Heisenbugs is to design the debugger to execute as a process separate from that of the debuggee's. Both of these will be discussed in detail.

An Overview of Debugger Features

There are many different services that a debugger can provide, more than can be adequately treated in a short paper. Some that we will not be discussing further are: graphical interfaces (some examples are the Joff debugger [4], Bugbane [5], Dbxtool [6], Pi [7], and pdb [8]); debugging optimized code ([9] is the classic reference); issues in source-level debugging [10] and symbol table management [6, 11, 12]; interpretive debuggers, such as Saber-C [13]; transfer tracing, where each discontinuity in the program counter is recorded [14]; debugging real-time software [15]; and multi-lingual debuggers [11, 16]. We also will not be discussing debugging of multi-threaded, parallel, or distributed programs (of which there is a substantial body of literature; [17] is a good starting place) except where those techniques also pertain to debugging single-threaded uniprocessor programs.

The debugger features we will be discussing further are:

- *single-stepping*: executing one instruction, source-level statement, or procedure of the debuggee at a time;
- *conditional breakpoints*: breakpoints that are triggered when reached only if a particular condition is true;
- *watchpoints*: "data" breakpoints that are triggered when a memory location is read or written;

- *teledebugging*: debugging programs remotely, with the (main part of) the debugger executing on one host and the debuggee executing on another;
- *attachment*: the ability to begin debugging a process that is already executing;
- *meta-debugging*: debuggers that can be used to debug themselves;
- *kernel debugging*: the ability to debug the operating system;
- *checkpointing*: taking a snapshot of the complete state of an executing program; and,
- *reverse execution*: the ability to roll back execution to a previous program state.

Each of these features can prove invaluable for certain types of debugging tasks. As we discuss different forms of support for breakpoint debugging, we will comment on how the support can be used for these features.

Breakpoint Mechanisms

Breakpoints provide a fundamental type of support for debugging. They give a debugger the ability to suspended the debuggee when its thread of control reaches a particular point. The program's call stack and data values can then be examined, data values possibly modified, and program execution continued until the program encounters another breakpoint location, faults, or terminates.

Breakpoints are usually implemented by replacing the instruction (or instructions) at the desired location in the program with a trap instruction[3]. The replaced instruction is remembered for later execution.

When the trap instruction is executed it causes a fault that is detected by the operating system. The operating system them informs the debugger (either via a message, by completing a system call that the debugger is executing, or by a straight transfer of program control) that a fault has occurred.

[3]Johnson [18] and Gondzio [19] give thorough discussions of different types and implementations of breakpointing and single-stepping.

The debugger inspects the type of fault and reports it to the programmer. At this point the programmer may wish to inspect or modify the debuggee's state. If the debugger shares its address space or part of its address space with the debuggee, this process is quite simple. If not, the debugger calls upon the operating system to provide a form of access to the debuggee's state.

When the programmer decides to continue execution, the program must be resumed starting at the instruction that was replaced by the trap instruction. Some systems provide a service to emulate the instruction; with others, the debugger must replace the trap instruction with the remembered instruction, single-step the debuggee, set the trap instruction again, and finally continue the debuggee's execution.

Conditional breakpoints are usually implemented using nonconditional breakpoints. Each time the debuggee stops at a breakpoint the debugger checks the condition. If it is false, the debugger continues the debuggee without informing the programmer. If it is true, the programmer is so informed.

When operating system support is required for detecting breakpoints and continuing debuggee execution, as it often is, this implementation of conditional breakpoints will have very low performance when the breakpoint is frequently encountered and the condition is usually false.

Instruction-level single-stepping is usually supported in hardware; we discuss such support below. Once a mechanism exists for instruction-level stepping, statement-level stepping can be implemented simply by repeated instruction-level stepping. Procedure-level stepping is more simply implemented by breakpointing the return address of the procedure.

Hardware Support

Hardware support for debugging comes in varying degrees. At a minimum, some form of trap instruction is necessary for implementing breakpoints, and virtually all processors provide this in some form. Beyond trap instructions, hardware can provide debugging support that has the major advantage of being non-intrusive: the debuggee runs wholly unmodified until the hardware detects a debugging-related event. This completely eliminates Heisenbugs.

Most processors support single-stepping and instruction emulation by providing a bit in the PSW that indicates that a trap should be generated

after the next instruction is executed. On processors without such support, such as the SPARC and R2000 architectures, the debugger must be able to decode the instruction to be single-stepped, compute its successor (or possible successors), breakpoint the successor, continue the program, trap the resulting breakpoint, and remove the successor breakpoint [20].

Hardware page protection can be used to implement watchpoints by making pages read-only (to catch modifications) or inaccessible (to catch any form of access) [11, 21]. The author's experience using this feature with VAX DEBUG is that it works well. It is not clear from the literature why this approach is not used more commonly. Possibly the success of VAX DEBUG is due to Vax's small page size (512 bytes), which limits the number of false alarms that must be processed. On machines with larger page sizes the overhead may become too high.

One powerful form of hardware debugging support is the availability of comparators that trigger upon specific execution or memory access patterns and then record the present execution state [22]. Snow [23] describes using an external logic analyzer for capturing such software states. The general mechanism was in turn used to implement a debugger [24]. Breakpoints were implemented by using hardware comparators to trap when a particular address was about to be loaded into the program counter. Similarly, watchpoints were implemented to trap reading, writing, or the writing of a particular value to a given address. Their implementation suffered, though, from the availability of only two simultaneous breakpoints, and from the interrupt delay of the external analyzer causing the debuggee to actually be stopped several instructions later than the breakpoint address itself.

Tsai [15] describes a system that makes sophisticated use of a logic analyzer for detecting high-level events such as process creation, system calls, interrupts, shared memory access, synchronization between multiple processes, interprocess communication, I/O completion, and process termination. The events are detected by triggering upon access or execution of addresses related the events. Tsai's system records precise timing information for each event, to allow debugging of real-time programs.

Another form of hardware debugging support is an *instruction counter* [25]. Cargill and Locanthi describe such a counter as being loaded with a value that is decremented upon the execution of each instruction. When the counter reaches zero, an interrupt is generated. Instruction counters provide a simple way to profile code by determining the number of instructions exe-

cuted. They also provide support for conditional breakpoints: the debuggee can be periodically stopped and the breakpoint condition inspected. When the condition is found to be true, execution can be restarted and binary search on the number of instructions executed used to find the exact point where the breakpoint condition becomes true. In a similar vein, an instruction counter can be used for reverse execution by counting the number of instructions executed until a fault and then restarting and running for just a few instructions less, to achieve the state where the debuggee is about to fault but has not yet done so. Finally, an instruction counter can be used to estimate elapsed program execution time by assuming that execution time is linearly proportional to the number of instructions executed. The authors implemented a 32-bit instruction counter for the Motorola MC68020 using 6 chips, but unfortunately they give no solid information about how the implementation was used or integrated into a debugging facility[4].

While Cargill and Locanthi make a compelling argument that an instruction counter gives a considerable benefit for a low hardware cost, Mellor-Crummey and LeBlanc describe how an instruction counter can be implemented wholly in software [27]. Their key observation is that for implementing watchpoints and reverse execution an instruction *count* is not needed; instead, what is needed is simply a unique identification of each state the program has entered, along with a way to recover the state corresponding to the identification. The value they associate with each program state is the pair (*program counter*, *software instruction counter*), where *software instruction counter* is incremented at least each time the program makes a backward branch or a subroutine call.

They implement their software instruction counter by modifying the assembly code of the program to be monitored. At each target of a backward branch and at the beginning of each subroutine body they add code to increment the *software instruction counter* and test whether it has reached a predetermined value. If so, a trap is generated or a branch made to a handler.

By keeping the *software instruction counter* in a general register, the overhead of the counter is minimized. Their measured overhead on a Motorola MC68020 machine ranges from negligible ($< 0.1\%$) for an execution of the

[4]Ditzel, McLellan, and Berenbaum [26] report that the CRISP microprocessor has a built-in timer/instruction counter, but also do not elaborate on the uses to which it has been put.

lex program up to 12% for an execution of *grep* with a particular pattern. Unfortunately, they predict that the overhead on a RISC processor will range from 15-23%, depending on the processor's instruction set. This overhead, combined with the necessity for special compilation to use the instruction counter, argues in favor of a hardware implementation.

The forms of hardware support for debugging considered so far are relatively cheap to implement. A much more expensive but powerful form of debugging support can be attained by using a *tagged* architecture, where each memory location has some type information associated with it. Johnson's SPAM architecture [18] uses tagging to implement watchpoints, run-time type checking, and catching access to uninitialized data, as well as many types of breakpoints (trap before instruction execution, after instruction execution, before successful branch, and on procedure entry and exit).

Another expensive form of debugging support is *sheaved memory*[28]. This is a method of grouping together a number of physical pages into one logical page. Writes to the logical page result in each member of the group being modified; reads come from the *primary* member of the group. Sheaved memory supports checkpointing and reverse execution. To create a checkpoint, one page is simply removed from each group. To reexecute at a given point the pages corresponding to the given checkpoint are copied to the primary member (and to any other members currently in the group) and execution is restarted. Sheaved memory systems are clearly memory-intensive when used for checkpointing, but if integrated with a copy-on-write paging system, perhaps no more so than other checkpointing schemes (see below). When checkpointing is not in use then each physical page is available for separate use; thus, the feature does not cost any additional memory except when used, only some additional hardware logic and operating system support.

While hardware support can clearly yield substantial benefits in debugger support[5], if present trends towards RISC-style architectures continue, debugger support will probably remain spare—a trap instruction and perhaps a PSW trace bit—since most hardware-supported debugging features can, with some effort and loss of performance, be done entirely in software. The fundamental tenet of RISC—that every feature supported by hardware

[5]It also can clearly be carried too far: Johnson mentions that proposals have been made for hardware support for mapping the program location to the corresponding source-language location!

be thoroughly justified—appears to require a greater emphasis on the importance of debugging than is now made.

Same-Process Debuggers

A major issue in debugger design is whether the debugger executes in the context of the same process as the debuggee or as a separate process (possibly on a remote machine). Both approaches are full of possibilities and limitations for debugging support. We look at each in turn.

A same-process debugger is one that executes in the context of the same process as the debuggee. Consequently, the debugger also shares the debuggee's memory.

The main advantages of same-process debuggers are:

- they can directly access the debuggee's memory, making both setting breakpoints and examination and modification of the debuggee's state a simple matter. This eliminates the need for operating system support to provide these features, which can result in substantially better performance (see below).

- the "context switch" from the debuggee to the debugger or vice versa can be extremely fast—as quick as a branch instruction. This can be especially valuable when faults are frequent, such as with programs that use floating-point emulation, since the performance degradation is much lower than when using a separate-process debugger.

There are, however, some serious disadvantages to same-process debuggers:

- The debugger's presence deprives the debuggee of some of its resources (such as virtual memory or open files).

- A bug that causes the debuggee to scribble on arbitrary memory can wipe out the debugger, rendering it useless for finding the bug.

- Same-process debuggers cannot be attached to already-running processes, nor can they support kernel debugging, teledebugging, or metadebugging.

- Finally, same-process debuggers are susceptible to Heisenbugs.

The modern examples of same-process debuggers tend to be those that run on microprocessors, such as Farley and Thompson's cdb debugger [29], designed to run on the DEC Rainbow 100, and Gondzio's MD-86 [19], designed for the Intel 8086. Workstation and minicomputer debuggers tend to be separate-process, no doubt because operating system support is available for separate-process debugging. One well-known same-process debugger that runs on minicomputers, however, is VAX DEBUG [11], which provides a large number of features, including multilingual debugging, source-level debugging, and watchpoints.

As is common with same-process debuggers, VAX DEBUG is implemented as a collection of exception handlers. When an exception is detected by the VAX hardware, the operating system first checks whether the faulting process has registered a primary exception handler. If so, it transfers control to the handler; otherwise, it walks up through the call stack looking for an exception handler. Provisions are made for registering additional exception handlers in case the stack walk fails to find one or encounters a corrupted stack frame. VAX DEBUG registers such exception handlers and thereby catches all program faults.

Separate-Process Debuggers

Most debuggers are designed as processes separate from the debuggee. The challenge with separate-process debuggers is keeping the overhead of manipulating the debuggee low.

Usually such manipulation is done via operating system calls.[6] On Unix systems, *ptrace* is used. This grab-bag system call provides mechanisms for processes to declare themselves as candidates for debugging as well as allowing a debugger to read and modify the debuggee's memory and access

[6] A wholly different approach is to use the file system instead. The Mesa "world-swap" debugger [30] works by utilizing a low-level system service that creates a file containing a (restartable) image of the debuggee's complete state. The debugger can then inspect and modify the debuggee's state by simply using file system operations. The chief drawback of this approach is that, as implemented, each world-swap takes tens of seconds to switch between the debuggee and the debugger.

additional state (such as registers and pending signals)[7]. A major flaw in the *ptrace* design is that all access and modification of debuggee state occurs one word or register at a time, and is done in the debuggee's context, so one system call plus two context switches are required for each word accessed.

Adams and Muchnick [6] found that *ptrace* system call overhead was a serious bottleneck in the performance of *dbx*; by extending *ptrace* to provide more bulk services (reading and writing of multiple data words and registers) they were able to reduce the number of *ptrace* calls by 2/3. Linton, the author of *dbx*, and Olsson et. al. also stress the need to reduce the number of system calls and context switches in order to achieve good debugger response [2, 31].

Context switches can be greatly reduced by making the debuggee's state directly available via system calls. For example, the **/proc** mechanism supported in System V Release 4 Unix systems [32] provides a virtual file system directory in which there is a file for each process. The "contents" of the file are the process's memory and system state. A particular part of the state can be read or modified by seeking to the appropriate address (offset) within the file and using the usual read and write system calls. No context switches are involved, only system calls. While **/proc** enhances performance and simplifies access to the debuggee's state, the system call overhead may well prove too high for state-access-intensive operations such as conditional breakpoints.

Teledebugging

When it becomes apparent that the debugger need not share the same process as the debuggee, the natural next step in separating the debugger from the debuggee is to provide support for remote execution of the debugger. With most separate-process debuggers it is the operating system that provides the necessary services for controlling the debuggee and accessing its state. If such system services are in general available to remote processes then it is straight-forward to write a remote debugger.

Unix does not make its system services available remotely, but other operating systems do. Mach, for example, provides a rather general exception facility [33]. The philosophy behind the design is that exception handling falls into two major classes: error handlers and debuggers[8]. Error handlers

[7] *ptrace* has been extended to provide for attachment, single-stepping, setting watchpoints, and accessing a process' symbol table: see, for example, [20, 21].

[8] A third class is that of emulators for instructions that are not present in the hardware

deal with low-level, correctable exceptions that may be unusual but are not necessarily due to bugs (for example, arithmetic conditions such as underflow). Mach provides a mechanism for registering an exception handler for either a thread (a particular thread of control within an address space) or a task (all threads of control within an address space). The former is intended for error handlers and the latter for debuggers.

When a thread generates an exception, it is trapped by the operating system and in turn forwarded to the exception handler for the particular thread, if any. If no handler has been registered, the exception is forwarded to the task's exception handler. Thus in some respects the Mach facility views debuggers simply as exception handlers, similar to same-process debuggers. There are some crucial differences, however:

- Because the registry mechanism uses the general Mach "port" facility, the debugger need not execute on the same machine as the debuggee. The port facility automatically forwards messages via the network if the destination process executes remotely.

- Unlike with same-process debuggers, Mach debuggers must rely on system services for manipulating the debuggee. These include services to get or set a thread or task's state and to read and write another task's address space.

- Because "attaching" a debugger to a task simply involves the debugger registering itself as an exception handler for the task, Mach debuggers may trivially attach themselves to already-executing tasks.

Caswell and Black [34] describe a version of GDB, a *ptrace*-based debugger [35], that has been modified to use the Mach exception facility. Their paper emphasizes issues concerning debugging multithreaded applications.

Another debugger that can be used for teledebugging is the Amoeba debugger [36]. The Amoeba debugger is "kernel independent" in the sense that no special kernel hooks are required for it—it is implemented using the more general mechanisms already provided by the operating system. The Amoeba debugger supports event-stream debugging as well as breakpoint debugging. Event streams are generated by linking the debuggee with a special run-time library that replaces the system call stubs with routines that generate events

instruction set. These are similar to error handlers.

reporting the system call. These event streams can be routed to an arbitrary destination (i.e., a local or remote debugger) using the standard Amoeba message facilities. The event streams are then filtered using finite state machines to find events of interest. Interesting events can then be logged and later used with a general Amoeba facility for checkpointing and restarting to support reverse execution. Unfortunately the paper does not give any details as to how the exceptions caused by breakpoints are communicated to the remote debugger nor how the remote debugger manipulates the debuggee's state, nor does it give references for these features. One assumes from the general introduction in the paper that these facilities are implemented in a fashion similar to Mach: an exception facility that can deliver exceptions to an arbitrary message recipient, and a capability-based set of system calls for manipulating a program's state.

Local Agents

While one of the primary goals in the Amoeba debugger design was that it be "kernel independent", and with Mach that debugging be incidentally supported by a more general facility, another approach to designing teledebuggers is by having a particular *local agent* that provides local (to the debuggee) support for a remote debugger.

An early example of this type of debugger is *Joff* [12], a debugger for programs running on the Blit programmable bitmap terminal [37][9]. The bulk of the debugger runs on the remote host to which the Blit is connected. The local agent is a lightweight process running on the Blit that has immediate access to the debuggee's address space[10]. Since the agent shares resources with the debuggee, care was taken to limit the resources required by the agent to minimize impact on the debuggee and related Heisenbugs. This required using fixed data structures instead of dynamic ones (for example, a maximum of 32 breakpoints is supported) and resulted in a basic design decision to do as much work as possible on the remote host. Because most of the work is done remotely, communication between the remote host and the local agent is via a simple, low-level protocol, with the remote host as master and the

[9]Note that *Joff* debugs programs that are running *on the Blit itself*; it is *not* a general-purpose debugger that simply uses the Blit for its user-interface.

[10]The Blit operating system is simple and permissive enough that the agent requires no special privileges.

local agent as slave. Because the protocol is so low-level (a typical request might be to fetch the contents of one long word at a particular location), complex debugging interactions such as conditional breakpoints require a great deal of interaction between the host and the agent, with consequent poor performance.

In some sense the Mesa "world-swap" debugger described above also makes use of a local agent, namely the low-level system service that writes the checkpoint images to disk. Because this system service is fairly independent of the rest of the operating system, it makes operating system debugging possible, too, since the service can be used to write a checkpoint image of the operating system itself. It is not difficult to envision modifying this service to support remote access in lieu of writing out the checkpoint image, for greatly improved performance, and this was done in the subsequent Cedar system, as (briefly) described in [38].

A more recent example of a debugger using a local agent is the Topaz TeleDebug (TTD) facility, described by Redell in [39]. Among the design goals for TTD were that it support remote debugging, that all levels of software from the interrupt and virtual memory systems on up to user-level applications be debuggable with the same debugger, and that the debugging system be resilient in the face of network and debugger crashes.

TTD consists of three components: the remote debugger, the TTD server (i.e., the local agent), and the TTD protocol, used for communication between the two. In line with the design goals, the protocol uses UDP/IP datagrams and thus supports a high degree of remote access, and the server maintains all critical state (thread state and breakpoints) locally so if the remote debugger crashes or the network fails another remote debugger can be activated to continue the debugging session.

In order to support both low-level system debugging and high-level application debugging, two versions of the TTD server were needed[11]. One version of the server, LowTTD, operates at an extremely low level, unable to use the memory management, scheduling facilities, or interrupt services provided by the operating system. HighTTD resides at the same level of the system as the operating system. HighTTD must still do its own storage management and exception handling, but can use some system services such using (a fixed

[11]Redell gives an interesting discussion of how the final design of these versions emerged from experience with a seemingly reasonable but actually problematic initial design.

number of) internal threads, accessing the network driver[12], and having a set of "good samaritan" functions be performed on its behalf by other processes. While LowTTD is at such a low-level that only operating system internals are visible to it, HighTTD is at a high enough level that it can provide access to user processes (but not the operating system internals).

Both LowTTD and HighTTD use the same TTD protocol, which consists of commands for reading and writing memory (in up to 1024 byte chunks), retrieving and modifying process state, setting and clearing breakpoints, single-stepping, and conveying the TTD server state to a newly activated remote debugger so it can pick up where another debugger left off.

Local agents thus have three major benefits: they make kernel debugging possible, they make teledebugging simple, and they provide a natural way to modularize those aspects of an operating system that pertain to debugging.

Checkpointing and Reverse Execution

A very powerful debugging tool is the ability to inspect the debuggee's previous states and possibly restart execution from one of them. Making a snapshot of a program's state at a particular point in its execution is *checkpointing*; "undoing" execution by returning to a previous state is *reverse execution*. The utility of such a tool is nicely illustrated in Feldman and Brown's discussion of their IGOR reverse execution system [40]. They address the debugging problem of a programmer discovering that a supposedly acyclic list contains a cycle: one of the elements points to a previous element. To find the bug responsible, the programmer would like to ask the debugger the question, "When did this tree first get an illegal cycle?" To answer this question, the debugger needs both to know how to recognize an illegal cycle (which might be specified by a program fragment) and to inspect previous checkpoints of the debuggee's execution until it finds the first one in which the tree contains an illegal cycle. The cycle must then first have appeared between that checkpoint and the previous one. By restarting the program from the previous checkpoint, the exact point where the cycle was introduced can be found.

An early implementation of reverse execution was done by Zelkowitz [41],

[12]The network driver contains a special hack that recognizes debugging packets and passes them along to HighTTD.

who modified a PL/C compiler to generate code to maintain a trace table in which all assignments to variables were logged. The implementation suffered from performance problems, though: it appears from the article that execution could only be reversed up the current call stack, and not back to procedures previously invoked but now exited; code expansion was about 40%; and execution slowed down by a factor of 2.

These limitations seem fundamental to the approach of generating extra code to keep a "running checkpoint" in memory. The IGOR system, on the other hand, generates checkpoints by writing the program state to disk at fixed intervals[13]. The checkpoint includes those pages that have been modified since the last checkpoint along with the names, modes, and file pointers of all open files. The checkpoints can then be browsed, restarted with new functions dynamically bound to them, or executed by an object-code interpreter to find the exact point at which a bug occurs. The present IGOR implementation is a prototype, with no debugger integration, very high performance overhead (40-380%), and very simple criteria for detecting the onset of bugs, but the authors make reasonable arguments that these hurdles are not insurmountable. The tool opens the door for easy pinpointing of a wide variety of previously hard-to-find bugs.

Such debugging tasks are very difficult to carry out using a conventional debugger. They involve repeated restarts of the program and a manual binary search to find the onset of the error. The search may be exceptionally tedious as it is often difficult for a programmer to know what point in a program's control flow represents approximately half way between two other points (an instruction counter would greatly aid the search).

If program execution includes some indeterminacy (due to parallel execution or real-time considerations), restarting the program may not reproduce the problem. The IGOR prototype cannot help with finding bugs in such programs. Not surprisingly, researchers interested in debugging parallel programs have devised a number of tools that provide for reverse execution in which indeterminacy is eliminated to varying degrees.

Instant Replay [42] works by recording the order in which locks are made for access to shared objects. It does not provide a checkpointing facility, but by enforcing the locking order upon reexecution a degree of reproducibility is

[13]The IGOR authors added a *ualarm* system call that generates a signal after a specified quantity of user CPU time has elapsed.

obtained. Bugnet [43] logs each interprocess communication (IPC) message along with a timestamp. It assumes that IPC is all done with system provided services, and thus cannot deal with shared memory IPC. A global checkpoint is made every 15 to 30 seconds, resulting in about a 2% performance impact. During replay, instead of actually delivering IPC messages, the contents of the message are retrieved from the log and delivered at the same time as before. The Bugnet author finds that the replay timing error is < 0.01 seconds, and conjectures that it would be even lower if the indeterminacies introduced by the paging system could be eliminated.

Support for State of the Art in Debugger Design

In this section we take a look at some existing or proposed debuggers incorporating state-of-the-art features and the support they require.

Recap is a state of the art reverse execution system now being built by Pan and Linton [44]. Their checkpointing design is wonderfully simple—just *fork!* The forked child then remains dormant; by definition, it holds the entire execution state as it existed at that moment. The debuggee can be replayed by sending a signal to the appropriate forked child, which first *fork*'s another copy of itself so the debuggee can be restarted at that point again, and then proceeds to continue execution. The efficiency of this approach naturally depends on the fork operation being implemented using copy-on-write; but as this is the trend in operating systems, the assumption appears well-founded.

Recap programs will link with a special run-time library that, as well as arranging for periodic checkpointing, logs the times of system calls and signals in order to eliminate these sources of indeterminism. The log is consulted upon playback to ensure the same order of execution as before. Recap will also include compiler hooks for generating code that logs access to shared memory. This log can then be used to enforce the same access ordering when reexecuting programs.

Recap promises to provide a comprehensive reverse execution facility, but relies on copy-on-write operating system support and compiler modifications.

The Parasight system [45] combines some of the ideas of same-process debugging and local agents. The system provides a mechanism for dynamically attaching *parasite* programs to a debuggee. These programs use shared memory facilities to access the debuggee's address space. *Parasites* run in

parallel with the debuggee and can be used to continually monitor the debuggee's state, looking for problems. In addition, the authors have implemented an intriguing *scan-point* facility. *Scan-points* are lightweight hooks that are patched into the running debuggee. When the *scan-point* is reached, control jumps to a procedure that has been dynamically linked into the debuggee. In this way, the programmer can write custom debug procedures, compile them, and dynamically "drop" them into a running program. The system includes a number of such "canned" procedures.

Parasight is now in the research prototype stage. It requires support for shared memory and dynamic linking. The main benefit of Parasight is that it provides very cheap yet protected debugging agents (the *parasites*) and a completely extensible (but not protected) mechanism for attaching arbitrary procedure calls to a running program. This latter facility can be used to provide extremely cheap conditional breakpoints.

Linton [2] briefly discusses the Ndb debugger, presently being designed by Pan. Ndb is a local agent-style debugger with a protocol similar to that used by TTD. The Ndb design focuses on the agent being a *server* that may provide access to a process for a variety of clients, including different debuggers and profilers. The use of a fixed Ndb protocol will allow Ndb clients to be written portably, without any knowledge of the particular implementation of the Ndb server. Thus, for example, an Ndb server could access a debuggee's memory in a safe fashion by mapping it into a portion of the server's address space, if the local operating system provides such support. If not, it will use whatever other debugging service that is provided by the operating system. A particularly intriguing aspect of Ndb is that it could be used in conjunction with a *compiler server*: a programmer could enter the code for a conditional breakpoint to the debugger, which would pass it along to the compiler server; when the compiler server returns the compiled code to the debugger, the debugger would pass the code to the Ndb server which would patch the code directly into the debuggee. In this fashion, very high performance conditional breakpoints could be supported.

One other area of debugger support where interesting work is on-going is that of "debugger languages" [1, 46]. These are languages used by a programmer to describe debugging commands and events-of-interest to a debugger. The most recent of these, Dalek [31], is event-oriented. Debugger- and user-defined events are held in event queues, from which the most recent or any previous event can be inspected. Debugging "programs" are written in

a combination dataflow (for events) and procedural (for actions) language; these programs then generate further events. Built in facilities such as a CPU time variable and "broadcast breakpoints" (breakpoints at every function whose name matches a regular expression) make it possible to write very powerful debugging programs such as profilers with a few lines of Dalek code. Another use of the facilities is for "on-the-fly code patching": the debugging language is rich enough that one can set a breakpoint at the site of a bug, have the debugger execute an interpreted version of the corrected code, and then branch around the faulty code upon restarting the debuggee.

Dalek is a prototype, built as an extension to GDB. It does not require any additional operating system support other than the *ptrace* call used by GDB, but since it hides much of the low-level interaction with the debuggee from the programmer, the programmer can easily write a debugging program that results in thrashing due to the high number of *ptrace* calls. The authors are presently considering reducing such overhead by switching to a **/proc** facility or to a method of using direct mapping of the debuggee's address space to access the debuggee's state, similar to the approach used by Parasight.

Summary and a Possible Future Direction

We have considered a number of themes concerning debugger support:

- Conditional breakpoints, though perhaps underappreciated, can provide powerful debugging support.
- Same-process debugging is valuable for performance reasons, but deficient due to resource-sharing, protection, and Heisenbug problems. It cannot be used for kernel debugging, remote debugging, attachment, or meta-debugging.
- Separate-process debugging is valuable for protection reasons, attachment, and meta-debugging. It suffers from performance problems.
- Teledebugging can be easily supported by an operating system that includes a network-transparent message service among its facilities, particularly if it supports remote operating system calls.
- Local agents provide a way to modularize the design of the entire debugging support system, as well as providing a natural means of supporting

kernel debugging. Like teledebugging, debugging with local agents can suffer performance problems if a great deal of IPC is required for communicating between the debugger and the agent (or operating system).

- Hardware support for debugging tends to be spare. Even debugging aids as simple as instruction counters, which can greatly simplify the implementation of reverse execution, are rare. RISC trends will probably keep hardware debugging support minimal.

- Reverse execution can be used for "time-travel" watchpoints and in debugging indeterminate programs. A particularly elegant implementation is to simply use the *fork* system service, providing copy-on-write is supported.

- Present work includes an emphasis on using local agents and shared memory to reduce debugging overhead, the ability to dynamically attach arbitrary code fragments to running programs to support conditional breakpointing, and investigations into powerful debugging languages to elevate the semantic level at which debugging is done.

Several of these themes might be gathered together in a debugger design as follows. We have argued that conditional breakpoints provide a very powerful debugging facility, yet in almost every debugger they are implemented by having the debugger evaluate the condition, resulting in at worst a large number of context switches and at best a considerable amount of debugger-agent message traffic as the debugger instructs the agent to fetch various words from the debuggee's memory for use in evaluating the condition. Ideally one would like the agent to perform the condition evaluation, since its access to the debuggee's state is much cheaper than the debugger's. Parasight attempts to do this by dynamically attaching procedures to the debuggee, but this scheme is slow, as it requires separate coding *and compilation* of the procedure, and is limited by the scope rules that the procedure must observe[14]. Furthermore, one would like to keep the agent's model of interacting with the debuggee as simple as possible, so one can support powerful debugging languages such as Dalek that are built up from simple debugging notions.

[14]For example, the procedure cannot inspect another procedure's local variables.

The problem is therefore how to effectively use the IPC channel between the debugger and the agent so that the agent can perform complex conditional breakpoint tests on behalf of the debugger. This problem is similar in structure to, for example, that faced by window system servers, in which one wishes to minimize the IPC traffic between the client (debugger) and the server (local agent) and yet have the server perform complicated actions on behalf of the client. An interesting design for solving this problem with window system servers is that used by NeWS—rather than using a particular, fixed protocol for communicating between client and server, the client sends *programs* over the IPC channel to the server, which then executes them on its behalf.

The proposed debugger design then is to extend the usual debugger-agent protocol to include a facility for downloading programs into the agent. The agent would then use a lightweight scan-point facility like Parasight's to breakpoint the debuggee. When the debuggee triggered the scan-point the agent would then execute the corresponding downloaded program to determine whether the breakpoint condition was satisfied.

To completely minimize the agent's overhead, the agent would run in the context of the debuggee. While this would make it susceptible to the resource-sharing, protection, and Heisenbug problems of same-process debuggers, the effects could be minimized by keeping the agent as small as possible and providing mechanisms for loading the agent to an arbitrary location in memory, so the probability of the debuggee accessing the agent's memory is minimal[15]. If the operating system supports dynamic linking then the agent can be loaded into running programs, supporting attachment.

Once loaded, the agent opens a user-level IPC connection with the debugger. When the programmer wishes to set a breakpoint, the debugger sends a message to the agent specifying where the breakpoint should be placed. The agent then patches that location to branch into the agent's code, such as is done by Parasight. When the debuggee executes the breakpoint, the agent immediately gains control and sends a breakpoint message to the debugger.

To implement conditional breakpoints, the debugger compiles the programmer's condition into a program and downloads it to the agent. When the debuggee executes a breakpoint, the agent executes the corresponding

[15]This relocation requires operating system support for non-contiguous virtual address spaces.

program and sends a breakpoint message to the debugger if the program so indicates; otherwise, it jumps back into the debuggee's code. The initial design would be for the programmer's condition to be compiled into an intermediate form, that would then be *interpreted* by the agent; this limits the complexity of the debugger, preventing it from needing to know how to generate machine code for the debuggee's processor. If this design proved too inefficient, the debugger could be extended to download fully-compiled code. In either case, the conditional code would execute in the context of the breakpointed procedure—its local variables and registers would all be available to the conditional breakpoint code—and would have the necessary low-level support for walking further up the execution stack to access other variables and inspect calling patterns.

The agent could also handle checkpointing requests by making *fork* system calls, and watchpoints by adjusting its (and, hence, the debuggee's) page protection and registering an access-violation handler.

This design provides for extremely fast "context-switching" between the debuggee and the agent; uninhibited access to the internal state of the debuggee, unlike with Parasight; fast translation between a condition the programmer wishes to explore and the subsequent "hooks" being ready in the debuggee, unlike Ndb or Parasight, which require separate compilation phases; and hooks for watchpoints and reverse execution. Finally, by writing a low-level version of the agent, support for kernel debugging could also be included.

References

[1] Bernd Bruegge. *Adaptability and Portability of Symbolic Debuggers.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, September 1985. CMU-CS-85-174.

[2] Mark A. Linton. The evolution of dbx. In *Proceedings of the 1990 Usenix Summer Conference, Anaheim, CA*, June 1990.

[3] W.C. Gramlich. Debugging methodology (session summary). In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* [50], pages 1–3.

[4] Thomas A. Cargill. The Blit debugger (preliminary draft). In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* [50], pages 190–200.

[5] Warren Teitelman. The Cedar programming environment: A midterm report and examination. Technical report, Xerox Corporation, Palo Alto Research Center, June 1984. CSL-83-11.

[6] Evan Adams and Steven S. Muchnick. Dbxtool: A window-based symbolic debugger for Sun workstations. *Software–Practice and Experience*, 16(7):653–669, July 1986.

[7] Thomas A. Cargill. Pi: A case study in object-oriented programming. In *Proceedings of the Usenix C++ Workshop, Santa Fe, NM*, pages 282–303, November 1987.

[8] Paul Maybee. pdb: A network oriented symbolic debugger. In *Proceedings of the 1990 Usenix Winter Conference, Washington, D.C.*, January 1990.

[9] John Hennessy. Symbolic debugging of optimized programs. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[10] John D. Johnson and Gary W. Kenney. Implementation issues for a source level symbolic debugger (extended abstract). In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* [50], pages 149–151.

[11] Bert Beander. Vax debug: An interactive, symbolic, multilingual debugger. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* [50], pages 173–179.

[12] Thomas A. Cargill. Implementation of the Blit debugger. *Software–Practice and Experience*, 15(2):153–168, February 1985.

[13] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C: An interpreter-based programming environment for the C language. In *Proceedings of the 1988 Usenix Summer Conference, San Francisco, CA*, June 1988.

[14] D. E. McLear, D. M. Scheibelhut, and E. Tammaru. Guidelines for creating a debuggable processor. In *Symposium on Architectural Support for Programming Languages and Operating Systems* [49].

[15] Jeffrey Tsai et. al. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8), August 1990.

[16] James R. Cardell. Multilingual debugging with the SWAT high-level debugger. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* [50].

[17] *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*. SIGPLAN Notices 24(1), January 1989.

[18] Mark Scott Johnson. Some requirements for architectural support of software debugging. In *Symposium on Architectural Support for Programming Languages and Operating Systems* [49], pages 140–148.

[19] Marek Gondzio. Microprocessor debugging techniques and their application in debugger design. *Software–Practice and Experience*, 17(3):215–226, March 1987.

[20] Sun Microsystems. *Ptrace(2), SunOS Reference Manual, Vol. II*, January 1990.

[21] Digital Equipment Corporation. *Ptrace(2), Ultrix documentation*, March 1990.

[22] W. Morven Gentleman and Henry Hoeksma. Hardware assisted high level debugging (preliminary draft). In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* [50].

[23] C.R. Snow. Integrated tools for hardware/software debugging, final report. Technical report, University of Newcastle upon Tyne, November 1987. Technical Report Series No. 247, S.E.R.C. Research Project GR/C/35974.

[24] W.Y.P. Wong and C.R. Snow. Implementation of an interactive remote source-level debugger for C programs. Technical report, University of Newcastle upon Tyne, January 1987. Technical Report Series No. 229.

[25] T.A. Cargill and B.N. Locanthi. Cheap hardware support for software debugging and profiling. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* [48].

[26] David R. Ditzel, Huber R. McLellan, and Alan Berenbaum. Design tradeoffs to support the C programming language in the CRISP microprocessor. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* [48].

[27] J. M. Mellor-Crummey and T. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* [47].

[28] Mark E. Staknis. Sheaved memory: Architectural support for state saving and restoration in paged systems. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* [47].

[29] Michael Farley and Trevor Thompson. A C source language debugger. In *Proceedings of the 1983 Usenix Summer Conference, Toronto, Ontario, Canada*, July 1983.

[30] R.E. Sweet. The Mesa programming environment. In *Proceedings of the ACM Symposium on Language Issues in Programming Environments*, pages 216–229. SIGPLAN Notices 20(7), July 1985.

[31] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, improved programmable debugger. In *Proceedings of the 1990 Usenix Summer Conference, Anaheim, CA*, June 1990.

[32] T.J. Killian. Processes as files. In *Proceedings of the 1984 Usenix Summer Conference, Salt Lake City, Utah*, June 1984.

[33] David L. Black et. al. The Mach exception handling facility. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* [17].

[34] Deborah Caswell and David Black. Implementing a Mach debugger for multithreaded applications. In *Proceedings of the 1990 Usenix Winter Conference, Washington, D.C.*, January 1990.

[35] R. M. Stallman. GDB manual (the GNU source-level debugger). Technical report, The Free Software Foundation, January 1989. Third Edition, GDB version 3.1.

[36] I.J.P. Elshoff. A distributed debugger for Amoeba. Technical report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, July 1988. Report CS-R8828.

[37] R. Pike. The Blit: a multiplexed bitmap terminal. *AT&T Bell Laboratories Technical Journal, Computing Science and Systems*, October 1984.

[38] D. Swinehart et. al. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[39] David D. Redell. Experience with Topaz teledebugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* [17].

[40] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* [17].

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

[42] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[43] Larry D. Wittie. Debugging distributed C programs by real time replay. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* [17].

[44] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* [17].

[45] Ziya Aral, Ilya Gertner, and Greg Schaffer. Efficient debugging primitives for multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* [47].

[46] Mark Scott Johnson. Dispel: A run-time debugging language. *Computer Languages*, 6(2), 1981.

[47] *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems.* SIGPLAN Notices 24(Special issue), May 1989.

[48] *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems.* SIGPLAN Notices 22(10), October 1987.

[49] *Symposium on Architectural Support for Programming Languages and Operating Systems.* SIGPLAN Notices 17(4), April 1982.

[50] *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging.* SIGPLAN Notices 18(8), August 1983.