Dependable Software Needs Pervasive Debugging

Timothy L. Harris
University of Cambridge Computer Laboratory
J J Thomson Avenue, Cambridge, UK
tim.harris@cl.cam.ac.uk

Abstract

Nobody would claim that debugging computer software is easy: all too often it proceeds by trial-and-error experiments in which programmers examine the behaviour of the system and form hypotheses that could explain what they see. These problems are exacerbated when developing distributed, peer-to-peer or multi-processor applications, or when unreliable network links form part of the system under test. Environments for pervasive computing take this to an extreme, allowing user-supplied code to run or migrate within and around the network.

In this paper we show how to perform pervasive debugging, enabling complex multi-process applications to be debugged and controlled as single entities and their robustness to changes in network performance to be evaluated. We do this by virtualizing the resources used by the system, allowing the threads that it involves and the network links that it uses to be modelled within a single controllable process.

1 Introduction

Debugging computer programs often proceeds by intuition: given some reports of how the system fails the programmer must try to deduce what might have led to those failures and then re-run the program in the hope of reproducing the same problem, or at least gaining a certain amount of corroborative or conflicting information. Eventually sufficient evidence may be gathered to focus the programmer's attention on the root cause of the problem.

This debugging process occurs even in simple projects – where the program is being run within a debugging tool, where it operates on a single computer or where it performs no external communication. This is lamentable: the system is deterministic and running in isolation, there should be no need to run and re-run it experimentally.

These problems multiply with the emergence of *pervasive computing* environments. For instance, in the Xenoserver project we are building an infrastructure for wide-area distributed computing [15]. We envisage a world in which Xenoserver platforms are scattered across the globe and are available for any member of the public to submit code. Grid computing and programmable networks present similar acute challenges [17]. Current methods for debugging distributed applications require the user to orchestrate separate tools attached to each of the processes involved; a tedious manual task.

In the remainder of this paper we first identify four key problems (P1-P4) faced by users of current debuggers and we then introduce the technique of *virtualized debugging* as a solution to these problems.

2 Challenges in Debugging

We believe that there are four particular problems that exist with conventional debugging tools:

P1. Stop/Inspect/Go Interface. The abstraction presented by a traditional debugger is of a controllable processor with support for stepping through the code, for setting breakpoints at which execution should halt and for inspecting (perhaps updating) the contents of memory locations in terms of the variables manipulated by the source code. However, the programmer must typically either set a breakpoint before a problem develops and step forwards (a slow process if the problem is erratic or its origin uncertain), or set a breakpoint when a problem is detected and examine the system's state to try to deduce how it reached that point. Such archaeology wastes programmers' time.

One solution is to generate extensive logging for off-line analysis. Larus' *whole program paths* [10], and the extensions proposed by Zhang and Gupta [19] represent the state of the art, recording the complete control-flow history of a single thread in a reasonably compact form. An alternative combines logging with *re-execution*, typically recording information during 'forwards' execution and using this to recover intermediate states during 'reverse' execution. The main problems are how to manage these logs efficiently, how to handle I/O and how to expose a natural *step backwards* operation to the user. In single-threaded systems Boothe's recent paper presents good approaches to all of these problems and extensive references to previous work [2].

P2. Risk of Masking Bugs. In shared-memory multiprocessor systems it is typical that different threads do not see memory access operations in a consistent total order – for example if one thread writes to location A and then to location B then a second thread may well read the new value of B and subsequently the old value of A. Such orderings might be caused by caching or write buffering within the processors, or by more advanced techniques such as value-based speculation. Adve and Gharachorloo provide a survey and tutorial of the subject [1]. The same problem exists within multi-threaded virtual machines, either through run-time code optimization or directly through the underlying processor [14].

Practical mechanisms for identifying bugs caused by these low-level problems have not progressed beyond asking experienced programmers to inspect the code. This straightforward approach can work well in a collaborative environment [7], but it does require an established and co-operative community of experts. Moreover, the uptake of SMP and SMT systems will accentuate the need for tools that support fine-grained concurrency, both within application code and within the OS.

P3. Poor Support for Concurrency. Co-operation is required but often lacking between the debugger and the thread system's implementation. For example the debugger may be aware of the threads managed by the operating system kernel, but be unaware of how application threads are multiplexed over each of these system threads (a problem we encountered using `dbx` under Tru64 UNIX). Conditional breakpoints must still be set in terms of the state of individual threads – not system-wide properties such as 'stop if Thread 1 holds lock A and Thread 2 holds lock B'. More generally operations such as single-stepping or resuming one thread do not take into account other threads in the system – for example whether they run freely, step forwards a similar amount or are suspended.

Deterministic replay schemes have been designed to allow consistent re-execution of multi-threaded processes. LeBlanc and Mellor-Crummey developed a system *Instant Replay* which, during forwards execution, logs the relative order of significant events by associating a version number with each shared object and tracking which threads access which versions [11]. For efficiency they assume that shared state is governed by multiple-reader single-writer locks. Choi and Srinivasan's work is typical of more general schemes in which significant events include accesses to shared memory and operations on mutual exclusion locks and condition variables [5].

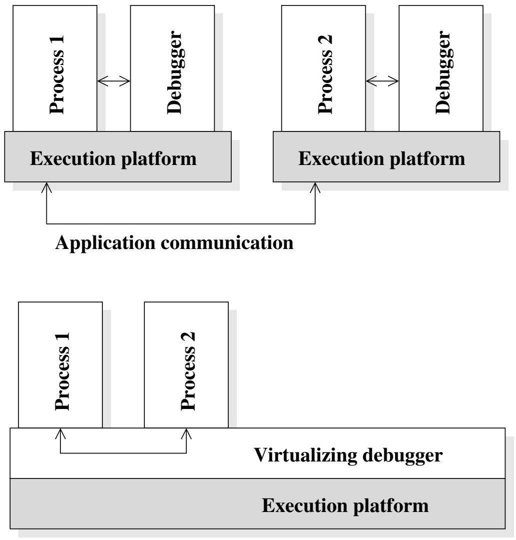

Figure 1. A traditional system (top) and a virtualizing debugger (bottom). Both are shown hosting a two-process distributed system. In the first case two separate machines and debuggers must be used; in the second case a single debugger provides two virtualized environments and communication occurs under its control.

A complementary approach is to identify classes of error automatically. The *Eraser* system is a dynamic data race detector [16] using binary rewriting to update per-location records of the set of locks protecting that address in the heap. Each time a location is accessed the associated lock set is intersected with the set of locks currently held: a warning is reported if the lock set becomes empty.

All these systems assume an interleaved model of single-process execution and so do not consider P2 or extensions to distributed systems.

P4. Poor Support for Distribution or Communication. Distributed programs pose even more substantial problems than those of multi-threaded or multi-processor systems: separate debuggers must be attached to the various processes involved. There is no central way to control system-wide parameters – e.g. to impose loss patterns or delays on communications, to corrupt or duplicate messages, to insert spurious ones or to control the relative execution speeds of threads. Such features must either be intrinsic in the platform running the tests or must be implemented as additional testing code by the programmer.

The Pilgrim debugger provides support for debugging distributed systems built using remote procedure call (RPC) [6], allowing call histories to be shown across machine boundaries. The *p2d2* distributed debugger provides a unified user interface to processes on different machines, and allows for interaction with e.g. MPI libraries [9]. We are not aware of any systems that attempt to support more general communication, for example at the level of TCP, UDP or indeed 'raw' sockets.

3 Virtualized Debugging

Each of the challenges identified in Section 2 arises because aspects of the system's behaviour depend on functions implemented outside the debugger's control. To address these problems we propose the technique of *virtualized debugging* in which all of the resources used by the system under test are virtualized by the debugger – ranging from low-level details such as the precise implementation of processor instructions to the scheduling of threads, the provision of separate virtual address spaces to different processes and network communication between those processes. Retaining control over the resources in use allows the debugger to ensure deterministic execution, to expand or contract the detail with which parts of the system are modelled, to manage distributed applications as single entities and to control the performance of external components such as communication links.

Conceptually, as shown in Figure 1, this places a single debugger below the entire system that it is being used to study, rather than having separate debuggers attached to each process. Brewer and Weihl suggested a similar structure for debugging high-performance parallel applications on a workstation-hosted simulator – we take their approach to an extreme by considering all resources and multiple machines [3]. In contrast to their processing-based environment, the execution of many distributed applications is dominated by communication latencies: programs being debugged may sometimes actually run faster under the debugger.

3.1 Scope

We envisage virtualized debugging being most useful in developing applications that use a moderate number of communicating threads or communicating processes – perhaps up to a dozen – each operating using the same instruction set and linked against the same libraries. Of course, allowing more heterogeneity would allow the system to have even broader applicability; but we see no shortage of problems to tackle given our assumed environment (either as constructors of the debugger, or as its eventual users).

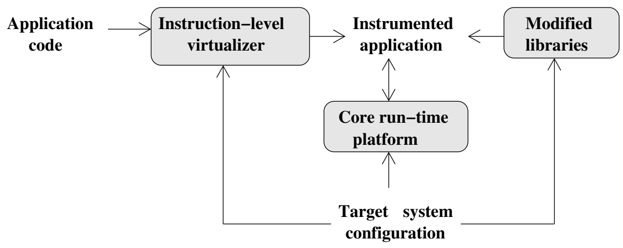

Figure 2. The interaction between the system components.

Hosting the entire system within a single debugger obviously imposes practical limits since it must be executed on a single machine. Raising these limits provides an avenue for future work, perhaps by re-distributing the virtualized environment. However, during our own work on lock-free data structures we have found diminishing returns, in terms of the number of bugs observed, going beyond four concurrent threads.

An interesting question is the extent that virtualized debugging can be used on code forming part of an OS. This is another 'classical' environment in which a lack of reproducable behaviour hinders methodical development – for instance the occurance of a deadlock may depend on the locks held by the application executing on the CPU to which an interrupt is delivered. Our current design will not support such an environment directly: although virtualized kernel-mode execution has been attempted elsewhere many times (for example recently by VMWare) incorporating it here raises a number of additional challenges. The first is the technical difficulty of virtualizing access to devices at the lowest level – that is, through I/O operations, DMA and the like. Secondly, the virtualizing debugger would require knowledge of the particular notions of 'thread' and 'process' that are used by the operating system it is hosting. We view the most direct application of virtualized debugging to OS implementation as being through the development of reliable libraries and modules from which the OS is constructed.

3.2 System Structure

The structure of the virtualizing debugger can be divided into three components: the *instruction-level virtualizer*, the *modified libraries* and the *core run-time platform*. The interaction of these is shown in Figure 2.

The *instruction-level virtualizer* is responsible for transforming the application code to allow it to be executed directly during debugging. For example by:

- Inserting periodic *yield* operations to return control to the main loop of the debugger (e.g. by using a software instruction counter [13]);

- Using sandboxing to check that memory accesses made by the application are to valid addresses [18]. This is necessary to prevent it from interfering with the enclosing debugger;

- Expanding accesses to shared memory locations into code sequences that emulate access-reordering, write-buffering or caching;

- Giving processes in a multi-process application the illusion of access to separate virtual address spaces, either by integrating operating system support or, more generally, by adding a per-process offset to each memory access made.

This transformation may either be performed ahead of time, or actually be performed dynamically on program 'hot spots' with a simple emulator used for uncommon code. External calls made by the resulting *instrumented application* are resolved against *modified libraries* – for example to emulate network access for communication between processes (perhaps introducing loss or delay). The modified libraries can also record the behaviour of I/O operations if reverse execution is to be supported. The *core run-time platform* is responsible for controlling the execution of the threads. During forwards execution, its main loop selects which thread should continue next and then executes that thread until it yields control.

Thread Scheduling. The implementation of the system requires care in order to avoid it masking further classes of bug. In particular, a simple regular placement of yield operations within the instrumented application risks limiting the execution schedules that could occur. Of course, the debugger could systematically explore all possible execution schedules around a program point; there is a clear trade-off between the number of schedules tested and the eventual coverage. However unlike traditional debugging the range of schedules could be changed dynamically, allowing the user to focus on specific parts of the program. In contrast, existing tools either require complete re-execution for each schedule, or perform exhaustive testing over entire runs (an impractical solution for non-trivial software) [4].

An alternative option is to ensure that the thread scheduling implemented by the virtualizing debugger remains typical of the behaviour that the real scheduler would provide. This could be achieved by having the run-time platform dynamically enable and disable a larger set of yield points so that, over a long execution run, thread switches would be considered at each possible location. If thread switches are based on a software instruction counter then the counter values that trigger switches could be drawn from a random distribution. A complementary approach would be to introduce a random (virtual) delay upon each thread switch to reduce the weak coupling effects that may lead to threads moving in lock-step [8].

Binary Re-writing. The concern of identifying access to 'external' resources is, of course, superficially similar to previous work on executing untrusted binary code – Wahbe *et al* introduce that area in their work on Software Fault Isolation (SFI) [18]. However, here we can benefit from closer integration between the debugger and the remainder of the tool chain – e.g. by introducing sandboxing checks before optimization rather than using simplistic binary rewriting, or by exploiting guarantees made by the language's type system.

Efficient Execution. The structure of distributed applications can be exploited to aid efficient execution over a virtualizing debugger: for instance, different nodes within a peer-to-peer system may run concurrently where allowed by the communications that they attempt. Similarly, if the instrumented application supports roll-back, then the core run-time platform may execute multiple threads from within the same process, check whether they did make conflicting memory accesses and, if they did, step back to before the conflict and then re-run sequentially. In each case the user is presented with the illusion of a single deterministic system whose execution they can control at all levels.

3.3 Dependable Systems

The availability of efficacious debugging tools does not in itself automatically lead to dependable computing systems. However, aside from the general practical benefits that virtualized debugging can bring to the software development cycle, there are extensions that could be of particular use for dependability. In particular, by exposing instrumentation and control interfaces for thread scheduling, it would be possible to express a variety of alternative tools using this same framework.

Firstly, simple coverage testing can be performed by recording each range of addresses that are executed and subsequently identifying code sections that have not been exercised. Secondly, for smaller programs, exhaustive testing is possible. This could be performed directly at a low level, testing each thread schedule in turn up to a specified depth or until a previously-observed state is reached. Alternatively, the programmer could define a mapping function from the concrete state of the system to a logical state and the search could explore that logical state space.

An important attraction of both techniques over symbolic model checking is that they operate using the same code that will ultimately be executed. While any exhaustive technique has limited scalability, integrating state-space exploration with the debugger could allow the programmer to initially run threads to a point at which problematic behaviour is observed and then use exhaustive exploration around that point to search for problems. Effectively this allows a limited test case to be generated directly from the full system, rather than requiring that it be extracted by hand.

4 Conclusion

Virtualized debugging provides an effective solution to the four problems identified in Section 2:

- P1 Arbitrary system states can be recovered by re-running the system from a previously recorded position. As with Boothe's work we can trade-off between the frequency with which checkpoints are taken and the time required to recover an intermediate point.
- P2 Arbitrary levels of processor detail can be included by the appropriate expansion of instructions. In addition to modelling memory accesses we could consider, for example, cache behaviours, TLB performance or the availability of specialized functional units.
- P3 The debugging interface has control over the scheduling policy and can therefore provide deterministic re-execution and selective stepping (or reversing) of specific threads.
- P4 By supporting multiple virtual address spaces within a single debugger it is possible to use the same framework with distributed applications. Breakpoints can be set according to system-wide properties and emulated communication links can be made subject to spurious transmission, to loss or to duplication.

In this paper we have outlined the technical infrastructure needed to support virtualized debugging: a further challenge is how to expose the facilities of this system to programmers. For example, in terms of the interfaces provided to control execution, or to set the delay and loss characteristics of links, or the thread and process scheduling policies.

Our implementation work is ongoing, building on techniques we developed for a currently-unpublished tool used to allow shared libraries containing static data to be used with our single-address-space Nemesis operating system [12] and on the open-source binary-instrumentation Valgrind tool. We hope to demonstrate a prototype at the Workshop in September 2002.

The delivery of dependable computer systems must be underpinned by the testing and evaluation of each of the components involved. As we have shown, existing debugging tools provide few of the facilities desired by today's programmers, let alone tomorrow's. Virtualized debugging provides a key remedy for this, applicable to a spectrum of settings ranging from distributed, peer-to-peer and agent based applications down to the implementation of concurrency primitives within a multi-processor OS.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[2] B. Boothe. Efficient algorithms for bidirectional debugging. In *Programming Language Design and Implementation (PLDI '00)*, volume 35(5) of *ACM SIGPLAN Notices*, pages 299–310, May 2000.

[3] E. A. Brewer and W. E. Weihl. Developing parallel applications using high-performance simulation. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 158–168, May 1993.

[4] D. Bruening. Systematic testing of multithreaded Java programs, 1999.

[5] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.

[6] R. C. B. Cooper. *Debugging concurrent and distributed programs*. PhD thesis, University of Cambridge Computer Laboratory, Feb. 1988. Also available as UCAM-CL-TR-128.

[7] J. Domingue and P. Mulholland. Fostering debugging communities on the Web. *Communications of the ACM*, 40(4):65–71, Apr. 1997.

[8] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *ACM Transactions on Networking*, 2(2):122–136, Apr. 1994.

[9] R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96)*, Philadelphia, PA, May 1996.

[10] J. R. Larus. Whole program paths. In *Programming Language Design and Implementation (PLDI '99)*, volume 34(5) of *ACM SIGPLAN Notices*, pages 259–269, May 1999.

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

[12] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, Sept. 1996.

[13] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '89)*, Apr. 1989. ACM SIGARCH Computer Architecture News 17(2):78–86, April 1989.

[14] W. Pugh. Fixing the Java memory model. *Proceedings of the ACM 1999 Conference on Java Grande*, pages 89–98, June 1999.

[15] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accounted execution of untrusted code. In *Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[17] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[18] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of 14th ACM SOSP*, pages 175–188, Dec. 1993.

[19] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Programming Language Design and Implementation (PLDI '01)*, volume 36(5) of *ACM SIGPLAN Notices*, pages 180–190, May 2001.