# PDB: Pervasive Debugging With Xen

Alex Ho and Steven Hand Computer Laboratory University of Cambridge, England first.last@cl.cam.ac.uk

## Abstract

Building distributed grid applications is notoriously difficult: the complex interactions between concurrently running processes, middleware, operating systems, underlying devices, and interconnecting networks can lead to unpredictable and difficult to analyze errors. Yet debugging support for such systems is woefully inadequate; typically a central user interface coordinates a set of conventional debuggers. This structure leads to synchronization problems and is limited to debugging user-mode applications.

In this paper we present the design and implementation of PDB, a pervasive debugger which executes in a virtualization layer underneath the entire distributed system. By running each node of a distributed application in a separate virtual environment atop the debugger, PDB can exercise full control over the entire execution environment.

## 1. Introduction

Developers often utilize parallelism for increased performance. By decomposing an application into a number of communicating components, the application can execute concurrently on a machine with multiple CPUs. Parallelism can be achieved with both data partitioning — running multiple copies of the same program logic concurrently on a different data subset — and functional decomposition — dividing the application into multiple functional units each running on a separate CPU. The recent development of grid computing extends the available parallelism further by distributing the application components across multiple machines connected by a network.

Grid applications or middleware using either form of decomposition are more difficult to debug than applications running on one machine. The interactions between independent processes must be checked. Both the data messages passed and the timing of those messages are important. The network between machines must be considered a Tim Harris Microsoft Research Ltd. Cambridge, England tharris@microsoft.com

non-deterministic black box which may delay or re-order messages.

Many current online debuggers for grid systems share a common architecture, comprising a central console that coordinates the activities of several independent debuggers. Debug code running on each remote node receives commands from the coordinator and then is responsible for interacting with one part of the target application.

Debuggers based on this architecture suffer from two basic limitations. First, their reliance on an underlying network makes it difficult to synchronize the activities of the coordinator and each remote debugger. Messages sent from the central control process cannot be guaranteed to be acted upon at remote machines at exactly the same time. The individual processes will stop at different points in their execution, thus preventing the debugger from generating a consistent global view of the entire distributed state.

Second, as a user-space application, each remote debugger is limited in its ability to analyze and alter the environment in which the target is running. The debugger, running as a peer to the target application, is constrained by the operating system's standard debugging interfaces. This visibility is generally restricted to the target's virtual address space and CPU state. It would be useful to extend this to deal with lower level events — e.g. to enter the debugger when a network packet is dropped, or when a particular set of processes are scheduled simultaneously.

#### 1.1. Pervasive Debugging

Pervasive debugging places the debugger in a virtualization layer above the hardware but beneath the target application and operating systems. These virtual environments can be implemented in several ways with differing emphasis on detail and performance. Hardware simulators, such as Bochs [14] for a single x86 CPU and Proteus [4] for a MIMD multi-processor, provide a highly accurate and detailed simulation of the inner workings of the CPU and other hardware components. They are useful for their ability to perform very low-level analysis, for example the impact of



pervasive debugging

Figure 1. Peer Debugging versus Pervasive Debugging. On the top, a traditional debugger is a user-level peer application of the target and thus is limited in scope and capability. Below, the pervasive debugger can analyze and affect the entire distributed computation.

changes to a CPU cache or re-ordering memory accesses. However, the use of detailed simulation comes at the cost of extremely poor performance.

Alternative virtual environments trade precision for greater performance. Para-virtualization provides a machine interface similar to the actual hardware. By requiring that an operating system be ported to run on a virtual machine monitor (VMM), a virtual environment can execute with minimal performance degradation compared to an operating system running directly on bare hardware [2].

This paper describes PDB, a pervasive debugger that utilizes the hardware and software abstractions provided by a VMM. Each component from the original distributed application runs in an isolated virtual machine that includes a virtual CPU, memory, network, and disk. Unlike a traditional application-level debugger, which executes as an adjunct process on each node, the pervasive debugger conceptually operates below the virtual machine in which the target application executes, as shown in Figure 1. The developer can control all aspects of the application and its environment.

One drawback of the application running on just one physical machine is that fewer system resources are available. It may be necessary to debug a smaller representative test case based on a subset of the original data. Alternatively one could extend the architecture to operate over several physical machines, as discussed in Section 5.

### 1.2. Horizontal and Vertical Debugging

PDB's unique placement enables the combination of two novel techniques, *horizontal debugging* and *vertical debugging*, when examining distributed systems.

Horizontal debugging allows a single debugger to control all of the nodes of a distributed computation. Heterogeneous distributed environments are supported; each virtual machine can run a different operating system. As there is no physical network between the debugger in the VMM and the target application on each node, PDB can synchronously examine concurrently executing processes.

This makes PDB ideal for examining interactions between the various components of the target distributed application. Instead of printf debugging or generating causal distributed breakpoints from different nodes based on Lamport clock time stamps [8], it is possible to stop the entire distributed application instantaneously. The developer can then analyze and verify program state either manually or with automated scripts.

Vertical debugging takes advantage of the hardware and software abstractions of the VMM to allow the programmer to examine and control the entire software environment including the operating system, system libraries, and virtual devices. With traditional debuggers, programmers are forced to treat the environment outside the application code that they have written as a black box and assume that they are correct. Unfortunately, this is not always true: an interface may be specified incorrectly, an implementation may be buggy, or a module may contain undocumented functionality. Any of these may lead to erroneous behavior.

Since PDB resides outside all virtual components, it can control the entire target application and virtual machine. For example, it is possible to step from application code through a system call into the operating system kernel and vice versa. A developer can also examine or modify network packets that have been transmitted by one node but not yet received by another.

This might help a developer writing a network application who is having trouble with packet loss. PDB can be used to follow packets to determine where the error is occurring. The debugger can successively examine the application network queue, operating system queue, and device queue to isolate the exact point where packets are lost.

The next section reviews existing debuggers for distributed systems. Section 3 describes the design of PDB, and Section 4 describes the current status of our implementation. Finally Section 5 discusses future work and concludes.

# 2. Related Work

Current debuggers for distributed systems can be characterized as *just a bunch of debuggers*: they share a common architecture comprised of a central console that coordinates the activities of independent debuggers on each node.

There is debug code running on each remote node interacting with the target. This code is either a default debugger supplied by the hardware vendor [19], a custom debug program [7], or a third-party debugger like GDB [6, 12]. In the case of heterogeneous distributed systems, the debugger may use of a variety of remote programs, each supporting a different target architecture [16]. This flexibility is achieved at the expense of providing only lowest common denominator functionality; any functionality not supported by every platform, such as the ability to examine a node's message queues [17], is lost.

Existing distributed debuggers may require source code changes to provide hooks and entry points. The debug processes have no way to intercede when the target application is initially invoked, because they run as peers to the target application. One solution requires special libraries with custom debug code for the distributed middleware to allow the debugger to intercede once the application has started [12].

Some debuggers use the ptrace system call or /proc interface [15]. These are essentially forms of interprocess communication that require context switches between the debugger, operating system, and target process and so can lead to poor overall performance.

An alternative mechanism is to add a new thread to the target process [1]. Running in the same address space but perhaps on a different processor, this debug thread can efficiently access the entire address space of the target. However, a malformed application can corrupt the debug thread.

Unpredictable communication delays between the master debugger and each remote debugger make it difficult to start or stop every participating node atomically [9]. Most debuggers message each node sequentially [20] or with a broadcast message [19]. This results in an imprecise operation; the nodes receive the messages at different times and any state aggregation will not reflect a consistent view. Another option employs a distributed snapshot algorithm [5]. Such algorithms require source code modifications to the target program to produce logical snapshots. These logical snapshots use causal event ordering to form a global state that only hypothetically may have existed; there is no guarantee that it actually did exist at any point.

Virtualized debugging [11] offers another approach to debugging multi-process applications. Various virtualization techniques are employed to control all of the resources utilized by the target application. The application code is interpreted [18] and runs with modified system libraries that emulate disk and network I/O operations. Thus, the virtual-



Figure 2. PDB Debugging a Grid Application. The unadulterated application runs in virtual machines atop a virtual machine monitor. The debugger is embedded within the virtual machine monitor and can control all aspects of the application and its environment.

ized debugger has fine-grain control over the application's execution environment; this interpret-everything model is suitable for a very detailed analysis of code execution and can uncover low-level bugs such as dependencies on CPU memory access ordering.

This extensive use of virtualization offers detailed precision at the expense of performance. The virtualized debugger runs as a traditional user-level application emulating the target environment. PDB, on the other hand, resides in a VMM below the operating system and supports native execution of the target application.

## 3. Design

Since PDB can control the application's virtual environment, its design is closely intertwined with the virtual machine monitor, as shown in Figure 2. The debugger is a client-server application. The PDB client is a user application that provides an interface for the user and sends commands to the PDB server. The server receives the commands and directs them to appropriate modules which interface with the virtual machine monitor. Additional functionality such as event processing is provided by specialized modules.

## 3.1. PDB Client

The PDB client provides a user interface that accepts commands from the user and translates them into instructions which are sent to the PDB server. Since the server only provides basic mechanisms for controlling the target, the client is responsible for various high-level tasks such as symbol table management and de-compilation.

## 3.2. Interface Protocol

The interface between the PDB client and server is based on the GDB remote serial protocol [10], which mandates basic functionality such as the ability to read and write memory and registers. This protocol has been extended to include a naming context and a security context to support pervasive debugging.

PDB can manage multiple threads in different address spaces; it is necessary to establish a target context for each command. Instead of passing just a *<virtual address>* to specify a particular memory location when installing a breakpoint, a *<virtual machine id, task id, thread id, virtual address>* tuple is required to fully qualify the desired address.

A security context is also required when the debug session is established. A particular developer's access can be restricted to only those processes comprising the target application. These may run in just a subset of the virtual machines currently executing. Multiple developers, each with a different context, can debug different applications concurrently without the ability to interfere with each other's work.

The debug protocol may execute over a network interface if the PDB client is running on a remote machine. Alternatively, the client can run within a local virtual machine and utilize a privileged debug interface to access PDB below.

### 3.3. Virtual Memory and CPU

PDB supports both breakpoints, which trigger when a particular instruction is reached, and watchpoints, which indicate when a particular memory location is changed. These can be implemented either in hardware using dedicated debug registers, such as DR0-3 on the x86, or in software, with the debugger automatically re-writing a particular instruction with another that generates a debug exception.

### 3.4. Virtual I/O Devices

With virtual I/O devices, the PDB can analyze data reaching beyond the operating system. For disks, the debugger can access and modify disk blocks directly. For networks, in-flight packets between virtual machines can be examined.

The debugger also extends the functionality provided by each virtual I/O device. For networks, a virtual topology can be created between the virtual machines. Arbitrary bandwidth, latency, and error characteristics can be assigned to each link between machines.

#### 3.5. Distributed Assertions

At certain points in the distributed computation, as defined by when a particular line of code is reached, the debugger can automatically check an assertion about the global state of the entire computation. The assertion is defined as a logical composition of a number of primitive events.

Two types of primitive events are supported: temporal events and spatial events. Temporal events, such as breakpoints, trigger when the instruction at a particular address is reached. Spatial events, which resemble watchpoints, check the contents of an object such as a memory address range, disk block, or network packet. The developer can check to see if a particular byte string occurs at a fixed location or anywhere within the target object.

## 3.6. Discussion

Placing PDB beneath the virtual environments enables two novel mechanisms for debugging distributed systems: horizontal debugging and vertical debugging. The debugger can synchronously control multiple applications running in different virtual machines and can examine the entire application environment beyond just the application code in question.

## 4. Implementation Status

The PDB server is being implemented within the Xen virtual machine monitor [2] for the x86 platform. Xen utilizes para-virtualization [21] to permit multiple guest operating systems to execute simultaneously, each in a protected x86 virtual machine (domain). Linux and NetBSD guest operating system ports to Xen have been completed; Windows XP and FreeBSD are in progress.

#### 4.1. Debugger Commands

The PDB client is currently an enhanced version of GDB [10]. Developers can run the client on either a remote machine or in a domain and connect to the debugger using the PDB interface protocol. The PDB server is event driven. Commands are received from the serial line, console keyboard driver, or via a Xen debug interface available to privileged virtual machines.

The debugger relies on GDB's ability to set soft breakpoints by re-writing an instruction with the one byte INT3 trap-to-debugger opcode. However, the PDB server must track the breakpoints set by the client. When a debug exception occurs the debugger first checks to see if it set the breakpoint. If not, then the exception is propagated back to the appropriate virtual machine for the guest operating system to handle; the exception was probably generated by a user-level debugger running within a virtual machine.

While the breakpoints could be saved as a tuple of <*domain id, process id, virtual address*>, it is more efficient to perform the <*domain id, process id*>  $\rightarrow$  <*page global directory*> translation once and subsequently cache the page global directory (PGD) value. Although in theory it is possible for an operating system to change a process' address space to another PGD, this does not occur in practice, and if necessary, the re-mapping could be caught and the cache invalidated.

#### 4.2. Reading and Writing Memory

*VMM Memory*. Every process maps Xen at the same virtual address. It is easy to access memory in Xen's address space using the current task; any PGD can be used. PDB can be used to analyze the VMM in which it is executing.

*Operating System Memory.* PDB fetches a virtual machine's PGD from a task structure maintained by Xen. Linear address translation is then used to find the appropriate page. Both three-level paging (4KB page size) and two-level paging (4MB page size) are supported, as Xen uses 4MB pages to minimize the number of L2 page table entries required. The physical address is then converted into an accessible virtual address.

*Process Memory.* Given a *<domain id, process id, virtual address>*, the domain id and process id combination are used to obtain a PGD. Operating system specific code is required for this step. Once the PGD has been fetched, linear address translation is used to access memory as in the above operating system case.

## 4.3. OS module

The debugger includes operating system specific modules in order to access specific data structures. For example, to find the PGD for a process in Linux, the code hashes the process id into the pidhash array and searches for the correct task\_struct. This is used to fetch the mm\_struct, or memory management structure, in order to retrieve the PGD for the particular process. Although this code presently resides within the virtual machine monitor, we plan to relocate it into the PDB client.

PDB currently supports Linux 2.4. Additional modules for FreeBSD and Windows XP are planned.

#### 4.4. Vertical Debugging

The debugger allows a developer to follow an application from user-space code into the operating system kernel. The developer can register interest when the target process enters or exits a system call.

During initialization, the guest operating system registers an INT 0x80 software interrupt with Xen. The exception is not installed in the CPU's IDT which causes the instruction, when executed, to generate a general protection fault. Xen handles the resultant interrupt and rewrites the entry point into the operating system with a software breakpoint. When this debug exception is reached, Xen can optionally set another software breakpoint to regain control with the system call has completed and the CPU return to user-level code.

## 5. Future Work

We are currently exploring advanced debugging tools which build upon our prototype. Whereas a traditional debugger operates on the contents of an applications virtual address space and CPU state, the pervasive debugger can incorporate the entire system state. By treating network packets and disk blocks as debugging objects, the debugger can execute actions based on them. For example, a conditional breakpoint could depend on a particular value within a network packet.

The VMM can also be used to implement bi-directional debugging. Most replay algorithms for distributed systems require custom debug libraries to log messages between nodes for deterministic replay. Serial applications, however, can take advantage of periodic checkpoints and repeated execution for increased performance [3]. This concept could be extended to a set of virtual machines. Xen offers two features that can be used to efficiently checkpoint the application and its environment: copy-on-write disks, and the ability to suspend and resume a virtual machine to/from disk.

PDB can inject faults to test an application's robustness to errors. The target system is perturbed in some fashion and the resulting behavior analyzed. Emulated faults can be introduced at many different levels, ranging from hardware errors to software programming errors. Hardware faults can occur within the disk, network, and memory subsystems. While these are difficult to reproduce without custom hardware, PDB can simulate their effects. Programming errors are generally simpler to generate by changing program state. PDB can also inject faults precisely. The debugger can monitor the distributed application and inject faults once a set of criteria, possibly defined on multiple nodes, is satisfied.

Similar to fault injection, directed execution can analyze a program's behavior. PDB can modify many aspects of a

program's execution environment in an effort to discover bugs. For example, stress testing the network may uncover bugs in an application's networking code. Assigning differing workloads to each virtual CPU can cause different deadlock situations to emerge.

Finally, we intend to investigate extending PDB to operate across a set of physical nodes while still retaining full control over the software under test. Using techniques borrowed from time-warp simulation [13], we can run the entire system optimistically in virtual time and use roll-back if nodes become out of sync. Nodes of the system would still be hosted within virtual machines; Xen's facilities for copy-on-write storage and tracking memory updates make this approach more feasible than if PDB were operating directly below processes.

This paper has described the PDB, a new method of debugging grid applications. By using a debugger incorporated into a virtual machine monitor, a developer gains the ability to vertically debug an application's entire execution environment. By mapping the distributed computation onto individual virtual machines, the developer can horizontally debug a number of process running in different operating systems.

## 6. Acknowledgements

This work is supported by EPSRC Grant GR/S63113/01 and by an Eclipse Innovation Grant from IBM.

## References

- Z. Aral, I. Gerther, and G. Schaffer. Efficient Debugging Primitives for Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 87–95, 1989.
- [2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium* on Operating Systems Principles, Oct. 2003.
- [3] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pages 299–310. ACM Press, 2000.
- [4] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT, Sept. 1991.
- [5] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems (TOCS), 3(1):63–75, Feb. 1985.
- [6] D. Cheng and R. Hood. A Portable Debugger for Parallel and Distributed Programs. In *Supercomputing*, pages 723– 732, 1994.

- [7] Etnus. TotalView. Product Brochure, Natick, Massachusetts, 2003.
- [8] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th International Conference* on Distributed Computing Systems (ICDCS), pages 134–141, 1990.
- [9] H. Garcia-Molina, F. Germano, Jr., and W. H. Kohler. Debugging a Distributed Computing System. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, Mar. 1984.
- [10] J. Gilmore and S. Shebs. *GDB Internals*. Free Software Foundation, Boston, Massachusetts, second edition, 2003.
- [11] T. Harris. Dependable Software Needs Pervasive Debugging. In Proceedings of the Tenth ACM SIGOPS European Workshop, Sept. 2002.
- [12] R. Hood and G. Jost. A Debugger for Computational Grid Applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, pages 262–270, 2000.
- [13] D. R. Jefferson. Virtual time. ACM Transactions on Programming Lanuages and Systems (TOPLAS), 7(3):404–425, 1985.
- [14] K. Lawton, B. Denney, N. D. Guarneri, V. Ruppert, and C. Bothamy. Bochs User Manual. 2003.
- [15] M. A. Linton. The Evolution of Dbx. In Proceedings of the 1990 Summer Usenix Technical Conference, pages 211–220, June 1990.
- [16] J. May and F. Berman. Panorama: A Portable, Extensible Parallel Debugger. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96– 106, Dec. 1993.
- [17] nCUBE. ndb man page, release 3.4.1 edition, Aug. 1994.
- [18] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.
- [19] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. A Scalable Debugger for Massively Parallel Message-Passing Programs. *IEEE Parallel & Distributed Technology: Systems & Applications [see also IEEE Concurrency]*, 2(2):50–56, Summer 1994.
- [20] P. E. Sorel, M. G. Fernandez, and S. Ghosh. A Dynamic Debugger for Asynchronous Distributed Algorithms. *IEEE Software*, 11(1):69–76, Jan. 1994.
- [21] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington Department of Computer Science and Engineering, Feb. 2002.