Distributed Source Code Debugging for Embedded Systems

Dale Parson, Luis Herrera-Bendezú and James Vollmer

Bell Laboratories / Lucent Technologies

**Proceedings of the 2000 International Conference on
Parallel and Distributed Processing Techniques and Applications**

Volume V, p. 2409-2415.

Computer Science Research, Education, and Applications Technology Press

Las Vegas, Nevada, June 26-29, 2000.

Distributed Source Code Debugging for Embedded Systems

Dale Parson
dparson@lucent.com
Bell Labs, Lucent Technologies
1247 South Cedar Crest Blvd.
Allentown, Pa. 18103

Luis Herrera-Bendezú
lherrera@lucent.com

James Vollmer
jvollmer@lucent.com

Keywords

embedded system, debugger, distributed processing, system on a chip

Abstract

Embedded system debuggers deal with several layers of target system abstraction, including circuit, machine code, assembly code, procedural code and extension code layers. The Luxdbg debugger contains an explicit API for each of these layers, providing opportunities for networked distribution at any of them. Having evolved in an environment of assembly code debugging for small-footprint signal processing systems, Luxdbg has concentrated on distributing the machine code layer of abstraction. Network efficiency is achieved by using a demand paging mechanism borrowed from operating systems. Paging minimizes network traffic and system call overhead in exchanging processor state between the debugger and its remote target processors. Luxdbg also uses an extension of the Composite Design Pattern to aggregate and to synchronize multiple processors and processes. A research version is adding networked distribution at the procedural code abstraction layer, based on the Java™ Debug Interface. It is also adding extension code distribution in support of multiple, communicating debuggers.

1. Introduction

Our group within Bell Labs designs and builds the LUxWORKS set of development tools for Lucent Microelectronics' embedded system processors. For most of the 1990's these processors consisted of fixed-point, assembly-coded digital signal processor (DSP) cores, together with application-specific peripherals and custom memory configurations. They have supplied signal processing capabilities in applications such as modems, wireless telephony and telephone answering devices.

We began building networked debugging capabilities in 1994, initially as a means of using TCP/IP to connect a UNIX-resident debugger to DSP debugging hardware housed in a PC. Our current embedded system debugger, Luxdbg [1], uses TCP/IP to provide distributed access to low-level processor data and control functions for one or more target embedded processors. Distributed access takes the form of fetches and stores of target processor state such as register values, memory contents and breakpoint triggers. Symbol-based interpretation of this state for C and assembly programs resides strictly within the debugger.

Ongoing application trends are increasing the need for distributed and concurrent debugging capabilities in Luxdbg. Older wireless systems configured their code statically, before deployment to the field, making it possible to perform fairly exhaustive testing and debugging at the development site. With the advent of distribution

infrastructure such as JINI™ [2] and Bluetooth™ [3] we anticipate a growth in demand for distributed debugging. Novel, untested combinations of software modules will arise dynamically, thanks to distributed, dynamic loading. Demand for concurrent debugging within embedded telephony systems is on the rise, both because of the bundling of microcontroller and DSP cores within a single silicon die for mobile units, and because of the synchronization of hundreds of DSPs in server systems such as cellular basestations.

Section 2 of this paper gives an overview of the layers of processor abstraction with which Luxdbg works. Section 3 shows how we have distributed one of these layers, the processor machine code layer, to support distributed debugging of small-footprint embedded systems. Section 4 outlines a design pattern-based approach to debugger synchronization of multiple processors that scales from embedded multiprocessors in a single chip to multiple processes distributed across the Internet. Section 5 examines distribution of other Luxdbg layers and discusses related work. Section 6 concludes.

2. Virtual machine debug interfaces

Figure 1 gives a schematic representation of the layers of abstraction in which Luxdbg debugging of an embedded application occurs. Each layer corresponds to an explicit Luxdbg application programming interface (API). Luxdbg users interact directly with the outer layer, and each layer provides access to portions of the layers supporting it.

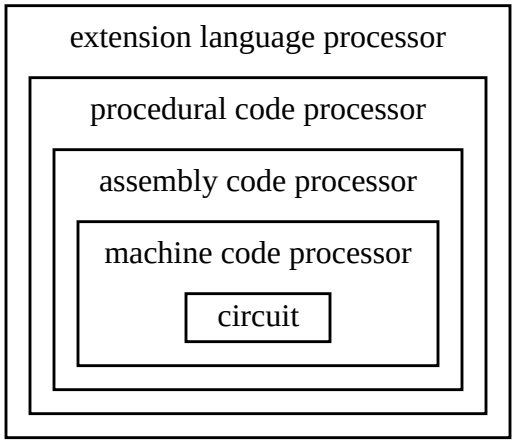

Figure 1: Layers of virtual machines

The *circuit layer* represents integrated circuit pins, registers, memory regions, peripheral devices and timing information. It may be embodied by C++ circuit modeling objects such as memory models in a processor simulation model, or it may be embodied by electronic circuits in a hardware processor. Luxdbg users can interactively read and write circuit scalar and vector values.

The *machine code layer* adds the abstractions of an *instruction stream* and a *system clock*. Artifacts such as an instruction pointer (a.k.a. program counter), program memory, hardware interrupts and breakpoints become evident at this level. With a system clock comes synchronization of multiple processor cores on a single chip.

The *assembly code layer* adds symbolic interpretation on top of programs running within the machine code abstraction. Unlike time-sharing systems, embedded systems typically do not carry much symbolic, source code information in the run-time environment. Often the run-time system is bereft even of a loader; programs then reside in ROM. Luxdbg's assembly layer adds a loader, symbol resolver and assembly expression evaluator to each machine being debugged.

The *procedural code layer* is a more powerful variant of the assembly code layer, adding constructs such as stack frames, data structures and objects that come with source languages such as C and C++. Both the assembly and procedural layers map symbolic commands to machine addresses and binary values.

The *extension language layer* refers to the existence of an embedded extension language within Luxdbg [4]. The production version of Luxdbg, written in C++, uses Tcl as its extension language [5, 6]. A Luxdbg user can write extension language *scripts* that interact with target processors to drive tests, to extend simulation models, and to synchronize execution of multiple processors from the debugger. The extension language adds a programming language interpreter to the set of commands provided by lower layers. With this layer a user can add additional, application-oriented debugging layers without requiring modifications to the underlying layers.

Any of the layers of Figure 1 could be distributed over a computer network. Distribution clearly puts the Luxdbg user on a system that is different from a target processor system. Dual-

system *cross-debugging* is typical for embedded systems, but Luxdbg also supports debugging a target system that is itself distributed across a network. The remainder of this paper looks at efficiency and synchronization issues in the distribution of these layers.

3. Distributed machine code layer

Distribution of the machine code layer of Luxdbg began in 1994 with its predecessor, an assembly-level debugger for Lucent's DSP1600 processor. Users could debug programs running within simulation models on both UNIX® and MS-DOS® computers. Unfortunately for UNIX users, debugging programs running on hardware required a PC-hosted interface card, necessitating moving to a PC. We distributed the machine code API of Figure 1 across TCP/IP using a custom remote procedure call (RPC) mechanism on top of sockets. WinSock sockets version 1.1 had recently arrived for Window 3.1® [7]; we built our own RPC because there was no viable commercial RPC package for connecting big-endian UNIX machines to little-endian PCs. Once we eliminated a major performance problem, UNIX users were free to avoid MS-DOS.

The performance problem came to light during testing of the distributed debugger. Debugger regression tests ran as much as 30 times slower when distributed over a local TCP/IP network than they did when executed stand-alone on an MS-DOS PC. Investigation uncovered a log-log relationship between the size of messages and their frequency. A typical regression test created on the order of 64K 8-byte messages and 1 64K-byte message. Small messages corresponded to assembly level reads and writes of individual registers and memory locations. They increased contention and collision rates on busy TCP/IP networks, and they increased the overall system call overhead within the debugger and hardware server. The few, large messages were less of a problem, but they still contributed communications overhead for blocks of data that the debugger did not always use. Each large message decomposed into several TCP/IP packets, with multiple socket system calls per message, and the debugger often used only a fraction of this data.

Figure 2 diagrams the mechanism that we prototyped in this earlier debugger and generalized in Luxdbg. Luxdbg maintains a machine code level *model* within the debugger process; this model is partially redundant with the state of the target processor. For the case of stand-alone simulation, where Luxdbg drives simulation activities, this model may be a complete *simulation model* capable of simulating execution. For the case of remote execution, on the other hand, the debugger's needs are met by a *state-bearing model*, a model that represents the state-bearing entities of the target processor, such as registers and memory, but which is not capable of simulation. A state-bearing model is essentially a *structured buffer.*

Luxdbg establishes initial contact with a remote processor by contacting a *target processor server* that is capable of serving multiple machine code *processors*. Our hardware server process, called TargetView™, provides debug access to multiple processors in a target system. TargetView

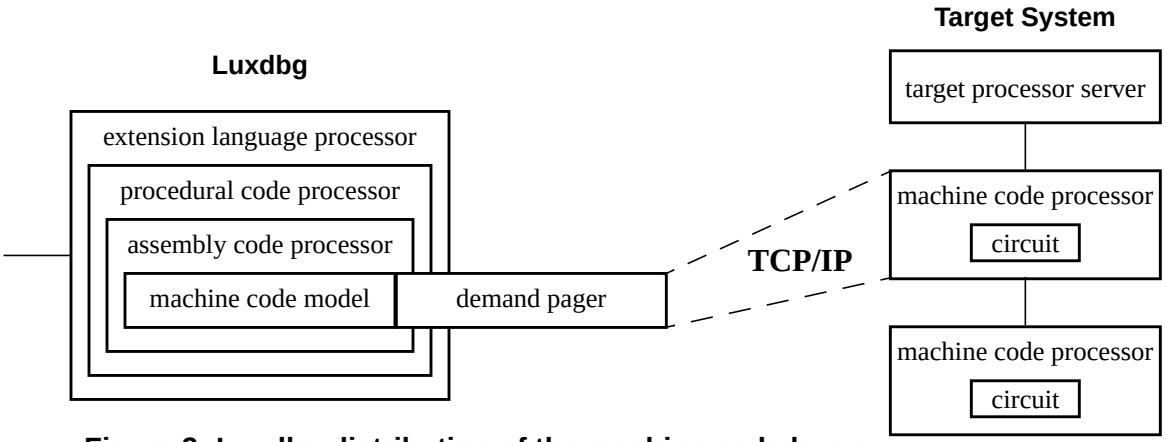

Figure 2: Luxdbg distribution of the machine code layer

manages connection setup, tear down, and device driver scheduling. Each debugger connection issues commands, queries and state changes to its target processors. The target server can also be a circuit simulator, where each target processor is a simulation model. In this case Luxdbg simulation models manage target TCP/IP connections and debugger interactions, avoiding coupling the off-the-shelf simulator to the Luxdbg debugger.

Once a connection is established, Luxdbg can either *download* the state of its model to the target environment, or *upload* the state of the target to its model. Download is useful for RAM-based programs; Luxdbg loads its *memory models* from an executable file, and then flushes these memory models to the target, thereby serving as a loader. Upload is useful for ROM-based programs that reside on a target embedded system; Luxdbg loads symbol table information from an object file, and matches the file's memory contents to the uploaded ROM contents to ensure consistency.

Demand paging operates when a target processor is stopped at a breakpoint and the state of Luxdbg's model of that processor is in agreement with the state of the target processor. User interaction occurs with the local model. Memory models are C++ objects with *fetch* and *store* operations. Every user store operation into a memory model object causes that object to mark the *page* of the stored location as *dirty.* Modification of nearby locations, for example fields in a contiguous data structure, modify only one or a few pages. When the user later invokes the *resume* command to resume processor execution, Luxdbg's model for that processor calls the *flush* method for each of that model's memory objects, and flush in turn calls a *pager* to write the dirty pages to the target.

After a target processor reaches a breakpoint, every debugger fetch invocation for a memory model checks the *resident* status of the page holding the fetch location. The aforementioned flush of model memory to its target system sets resident status to *remote*. When a user requests the contents of a location residing in a remote page, the model memory object obtains the necessary page from the target processor and marks its resident status as *local*. Subsequent fetches from that page do not entail communications with the target processor. A store to memory requires that the store location's page is local; after storing a value, the memory model marks the page as dirty. Processor resumption repeats the cycle.

Paging is transparent to the debugger user. The procedural and assembly code layers translate symbolic inspection and modification commands into memory fetches and stores, with any resulting memory paging hidden within memory models.

This is precisely the demand paging strategy that operating systems use, and it works here for essentially the same reason: *locality of reference*. Memory accesses by a Luxdbg user or Tcl extension script tend to cluster within a small number of pages per breakpoint. Paging avoids incurring a set of network request and reply packets for each storage access for nearby locations, and a reasonable page size avoids transferring unneeded memory blocks for a single socket message.

Empirical experimentation determined that an 8K page size works well for typical debugger usage. The impact on regression tests dropped from 30 times slower to 2 times slower for distributed debugging when compared to non-distributed debugging, and then only for worst case, automated regression tests with a high volume of debugger-to-processor memory interactions. Interactive user access to memory does not suffer from any noticeable delay.

Each memory model is a C++ memory object from the circuit level of abstraction, such as on-chip RAM as illustrated in Figure 3. Each processor model records the offsets of memory models in its linear address space, and each memory model records the processors and memory offsets at which it is visible. A memory model pages its contents to and from its remote execution environment, but paging to hardware may require execution of debugger code on the target system, and this code refers to a linear address space, not to a primitive block of memory. In Figure 3, for example, the block of shared RAM appears at one location for the DSP and at another location for the microcontroller. Each processor model includes a pager object that is specific to its processor. It translates a memory model's page location to a logical address in the linear data space of its target processor, and it communicates this address to the target. For Figure 3 either target processor could assist with paging shared RAM by communicating with the pager in its Luxdbg processor model.

The shared RAM block of Figure 3 provides

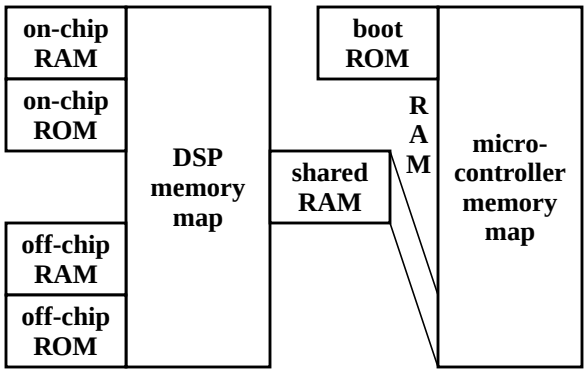

Figure 3: Shared memory mapping & paging

an example of parallel processor debugging in Luxdbg. Consider the case where multiple processors share a block of memory, and a C data structure in shared memory is being over-written by an invalid pointer-based assignment. A debugger user can set a breakpoint that triggers whenever the structure's range of locations is written by any processor; the breakpoint then identifies the processor. Both our processor hardware and simulation models include this capability. By conveying the processor ID to Luxdbg as part of the breakpoint information, Luxdbg is able to locate the offending processor and code quickly.

4. Multiprocessor synchronization

Luxdbg supports debugging of multiple processing cores within a system on a chip (SoC). Unlike most source code debuggers, Luxdbg interacts with two types of target processors — instruction streams and processor groups. An instruction stream processes a single sequence of instructions at some level of abstraction from Figure 1. A machine code instruction stream processes a sequence of binary opcodes and operands, and assembly and procedural symbol interpretation add symbolic layers.

A processor group, on the other hand, is an aggregation of instruction streams and nested processor groups used for synchronization. At the lowest, circuit level, a common master clock controls synchronization by driving all hardware processing cores within a chip. Aggregation appears at more abstract layers as well. Processor grouping provides the underlying class for extending the semantics of serial debugger commands to support multiprocessor debugging. Grouping allows commands to be applied to a set of processors instead of a single one at a time.

Figure 4 gives a UML class diagram for classes InstructionStream, ProcessorGroup, and their common base class, Processor. Processor is an abstract class with attributes such as processor state (running or halted), and operations such as start() and stop(), that are common to its derived classes. InstructionStream is a concrete class that adds instruction stream-specific attributes such as program counter and program memory, and instruction stream-specific implementations of operations. ProcessorGroup is a sibling of InstructionStream. It adds attributes and operations that allow it to collect instances of any Processor type as well as to synchronize the starting and stopping of their execution. The diamond-headed connection from ProcessorGroup to Processor signifies containment of zero or more Processors.

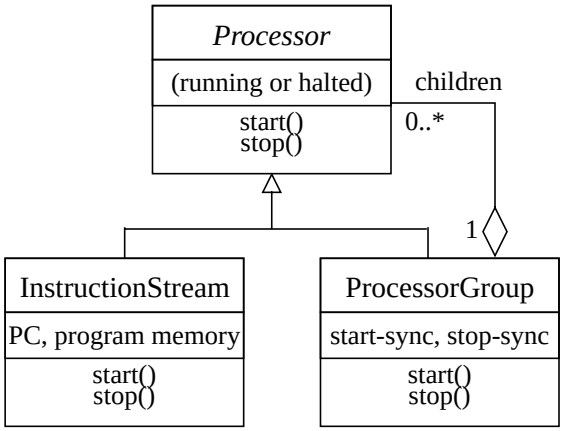

Figure 4: Classes of Luxdbg processors

Figure 4 is an example of the object-oriented Composite Design Pattern [8]. A benefit of this pattern is that it allows clients, in this case debugger users, to refer to atomic entities (InstructionStream) and aggregate entities (ProcessorGroup) in a uniform manner. A ProcessorGroup delegates execution commands such as start() and stop() to its members, and an InstructionStream executes these commands. Group operations applied to an InstructionStream do nothing or are delegated to a parent ProcessorGroup, depending on the operation. Delegation occurs when the InstructionStream belongs to a ProcessorGroup that enforces synchronization among its InstructionStream members. InstructionStream-only operations applied to a ProcessorGroup typically throw

exceptions. This pattern allows a user to issue commands without regard to processor type, and the receiving object reacts appropriately. Client code is simplified, and users view all objects to be debugged from a single perspective.

The classes of Figure 4 have many concrete realizations in the world of distributed computing. Examples include multiple processor cores within a system on a chip, threads within processes within an operating system, chips served by a hardware debug interface, processor simulation models within a circuit simulator, and computing nodes within the Internet itself.

These symmetries are more than aesthetic. Processor servers typically have ways of synchronizing contained processors at some level of temporal granularity. Luxbdg exploits this aspect of the Composite Pattern as applied to processors to achieve coarse and fine-grain synchronization.

Luxdbg distinguishes several subtypes of ProcessorGroup, including DebuggerGroups and ServerGroups. DebuggerGroups exist inside the debugger. A user can configure a DebuggerGroup to cause synchronous start or asynchronous start of its members, and independently, synchronous stop or asynchronous stop of its members. Starting one member of a synchronous start group starts execution for all members of that group, and stopping one member of a synchronous stop group (e.g., via a breakpoint) stops execution for all members of that group. Asynchronous groups start or stop members independently, and issuing a command at the level of a group itself acts as a synchronous command, affecting all members.

Default synchronization within a DebuggerGroup has coarse temporal granularity. Luxdbg issues “start” or “stop” commands for nested Processors in a sequence. There are no guarantees about timing. Nevertheless, the feature is useful for debugging concurrent processors or processes. Such processors or processes may share resources, and by using synchronous start and stop, a Luxdbg user guarantees that all processors reach quiescence at a breakpoint, avoiding contention between the debugger and processors for common resources such as shared memory.

Some run-time environments can support tighter synchronization, and this is where ServerGroup plays a role. ServerGroups exist in the target systems being debugged, in the form of operating systems, hardware servers such as TargetView, circuit simulators or CORBA ORBs. Luxdbg deals with a ServerGroup first when it establishes a connection to a processor. Luxdbg deals with a ServerGroup again when the latter is capable of *target-assisted synchronization*. In cases where Luxdbg determines via a query that a ServerGroup supports fine-grain synchronization, Luxdbg replaces its default, sequential start and stop actions for a DebuggerGroup with target-assisted synchronized start and stop for members of the DebuggerGroup served by that ServerGroup. In this way Luxdbg can take full advantage of the temporal characteristics of the target execution environment, without over-coupling itself to the details of a particular target environment.

5. Higher layers and related work

Most of Luxdbg’s networked distribution occurs at the machine code layer of Figure 1 as a result of the small-footprint nature of embedded, signal processing systems. However, Luxdbg has dealt with some degree of distribution of the other layers of Figure 1.

Distribution at the extension language level is relatively easy because extension language interpreters evaluate textual commands, and text is easy to pass among distributed processes. Luxdbg’s graphical user interface (GUI) passes Tcl command strings to the debugger process via a socket as its medium of communication. A research version of Luxdbg is looking more deeply at this form of distribution. It is building support for distributing multiple *extension language encapsulated debuggers* across a network, spawning debuggers at remote sites, close to their target processors. Debugger instances communicate by passing extension language commands and return values. Research Luxdbg treats each extension language encapsulated debugger as a ProcessorGroup, i.e., a collection of Processors that in this case can be manipulated, interactively, by a user or extension language script.

Distribution of assembly language and procedural layers of abstraction is common among other, non-embedded system debuggers that rely on the presence of an operating system and loader in the target environment. Debuggers such as gpdb [9] and cdb [10] create a debug agent that

executes within a target's operating system environment. The agent is similar to Luxdbg's target server processes, but it also has access to symbol table information for its target program. It is capable of mapping function and variable names to addresses. In gpdb it can replace entire function definitions at run time. Given target system loader and operating system support for symbol manipulation, this larger footprint approach is a viable alternative to Luxdbg's small footprint approach for embedded systems. As embedded systems come to include more substantial memory resources and operating systems, this approach may come to apply to embedded systems as well.

Another upcoming procedural/object level debug API is the Java™ Platform Debug Architecture [11]. The lowest level of this three-tier debug architecture is the Java Virtual Machine Debug Interface (JVMDI), a native C API that acts as a powerful reflection interface into a running Java Virtual Machine.The second layer, the Java Debug Wire Protocol (JDWP), defines a protocol that allows remote debuggers to invoke JVMDI operations. The top layer, the Java Debug Interface (JDI), is a Java API that operates atop JDWP. The research version of Luxdbg, written in Java, is adding a JDI interface as the basis for the procedural level API of Figure 1. We plan to derive a C++ debug API from JDI, by defining C++ semantic interpretations for JDI interfaces, and by adding C++-specific extensions where necessary. The result will be a generalized debug API for the procedural code layer that will work for multiple object-oriented languages.

6. Conclusion

The partitioning of Luxdbg into an architecture that makes a clean distinction between low-level, machine-oriented abstractions and high-level, symbol-oriented abstractions, lends itself well to multiple types of networked distribution. The machine code abstraction has worked particularly well for distributed debugging of small-footprint embedded systems. As embedded systems increase in complexity and resources, and as distribution of embedded systems becomes prevalent, Luxdbg should be well positioned to take advantage of distribution of its other conceptual layers.

7. References

1. D. Parson, P. Beatty, J. Glossner and B. Schlieder, "A Framework for Simulating Heterogeneous Virtual Processors." Los Alamitos, CA: IEEE Computer Society, *Proceedings of the 32nd Annual Simulation Symposium*, IEEE Computer Society / Society for Computer Simulation International, San Diego, CA, April, 1999, p. 58-67.
2. W. Keith Edwards, *Core JINI™*. Upper Saddle River, NJ: Prentice-Hall, 1999.
3. Ericsson / Bluetooth, http://bluetooth.ericsson.se/, April, 2000.
4. Dale Parson, "Using Java Reflection to Automate Extension Language Parsing." Berkeley, CA: USENIX, *Second Conference on Domain-Specific Languages Proceedings*, Austin, Texas, October 3-5, 1999, p. 67-80.
5. Brent Welch, *Practical Programming in Tcl and Tk*, Third Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1999.
6. D. Parson, P. Beatty and B. Schlieder, "A Tcl-based Self-configuring Embedded System Debugger." Berkeley, CA: USENIX, *The Fifth Annual Tcl/Tk Workshop '97 Proceedings*, Boston, MA, July 14-17, 1997, p. 131-138.
7. Ralph Davis, *Windows Network Programming*. Reading, MA: Addison-Wesley, 1993.
8. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
9. Kim Elms, "Debugging Optimised Code Using Function Interpretation." AADEBUG'97. *Proceedings of the Third International Workshop on Automatic Debugging*: Linkˆping, Sweden, May 26-27, 1997, http://www.ep.liu.se/ea/cis/1997/009/.
10. David R. Hanson, "A Machine-Independent Debugger — Revisited," *Software—Practice and Experience* 29(10) (August, 1999), p. 849-862.
11. Java™ Platform Debug Architecture, http://web2.java.sun.com/products/jpda/doc/, April, 2000.