Evolution in Kernel Debugging using Hardware Virtualization With Xen

Nitin A. Kamble
nitin.a.kamble@intel.com
Jun Nakajima
jun.nakajima@intel.com
Asit K. Mallick
asit.k.mallick@intel.com

Open Source Technology Center, Intel Corporation

Abstract

Xen’s ability to run unmodified guests with the virtualization available in hardware opens new doors of possibilities in the kernel debugging. Now it’s possible to debug the Linux kernel or any other PC operating system similar to debugging a user process in Linux. Since hardware virtualization in-processor enables Xen to implement full virtualization of a guest OS, there is no need to change the kernel in any way to debug it.

This paper demonstrates the new evolutionary debug techniques using examples. It also explains how the new technique actually works.

1 Introduction

The Xen[1] open source virtual machine monitor initially started with software virtualization by modifying the guest OS kernel. Since Xen 3.0, it also supports the Intel® Virtualization Technology® [2] to create and run unmodified guests. This Xen capability to run unmodified Linux OS or any other unmodified OS also provides a new opportunity to debug an unmodified OS using the Xen VMM.

With this guest debug capability, it is possible to trap into an unmodified guest such as any Linux, Windows, DOS, or any other PC OS; and check the register state, modify registers, set debug breakpoints anywhere including in the kernel, read and write memory, or inspect or modify the code currently being executed. This new method uses gdb[3] as the front end for debugging. With gdb also comes the source-level debugging of an unmodified Linux kernel. There are some advantages of using this debug approach compared to other kernel debug options, such as the Linux kernel stays unmodified, and ability of setting of breakpoints anywhere in the code. In fact it is also possible to set breakpoints in the boot loader such as grub [4] or inside the guest BIOS code.
The Architecture and Design of debugging of an unmodified guest

The virtualization technology in the processor, and Xen’s ability to take advantage of it, let an unmodified OS run inside a virtual machine.

The following sections first briefly describe the virtualization technology in the Intel IA32 processors, and how Xen hypervisor utilizes this hardware virtualization to create virtual machines (domain) for unmodified guests.

2.1 Intel Virtualization Technology for IA32 Architecture

Virtualization Technology in the Intel processors augment the IA32 architecture by providing new processor operation modes called VMX operations. And the Virtual-Machine Control Structure controls the operation of the virtual machine running in the VMX operation.

The following subsections introduce the VMX Operation and the Virtual-Machine Control Structure briefly.

2.1.1 Introduction to VMX operation

VT processor support for virtualization is provided by a new form of processor operation called VMX operation. There are two kinds of VMX operations: VMX root operation and VMX nonroot operation. The Xen VMM runs in VMX root operation and guest software runs in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits. Figure 1 depicts the interactions between the VMX root and VMX nonroot operations.

Processor behavior in VMX root operation is very much as it is outside VMX operation or without the VT feature in the processor. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited. Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

Because VMX operation places these restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed.

2.1.2 Virtual-Machine Control Structure

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual machine control structure (VMCS). Ac-
cess to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer can be read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using other instructions: VMREAD, VMWRITE, and VMCLEAR.


2.2 Xen support for unmodified Guest using the Hardware Virtualization

The processor state of the running vcpu is stored in the VMCS area. Xen uses a different VMCS for each unmodified guest vcpu. So when it is scheduling from one VMX guest to another VMX guest, it switches the VMCS to save and load the processor context automatically. To get into the hypervisor, paravirtualized Guests use hyper calls, similar to a process doing sys-call into OS for privileged operations. On Xen, unmodified guests run in restricted mode (VMX nonroot operation). In that mode all the virtualization-related processor instructions and events cause a VM Exit, switching to the hypervisor. With the VM Exits there is no need to modify the guest OS to add the hyper calls in the kernel.

The unmodified guest OS thinks that it is in control of its physical memory management, such as page tables, but the Xen hypervisor is monitoring the guest page table usage. Xen handles the page faults, TLB flush instructions for the Guest OS, and maintains shadow-translated page tables for the unmodified guests.

2.3 Internals of debugging an unmodified guest on Xen

Figure 3 shows the interactions happening in various Xen components when an unmodified guest is being debugged. Both gdb and gdbserver-xen are processes running in the Xen-paravirtualized service OS, also known as domain-0. gdb is totally unmodified. gdbserver is a gdb tool used for remote debug. gdbserver-xen is a modified gdbserver for utilizing Xen hyper call based interfaces available in domain-0.

The following sections describe interactions and implementation details for the the Xen components exercised while debugging a unmodified guest OS.

2.3.1 gdb and gdbserver-xen interactions

The gdbserver [7] is a standard tool available to use with gdb for remote gdb. It uses an ASCII protocol over the serial line or over the network to exchange debug information. See Figure 2 for a pictorial view of this interaction.

The gdbserver implements target_ops for Linux remote debugging. And gdbserver-xen basically extends the standard Linux gdbserver by implementing the target_ops specific to Xen. The interaction between gdb and gdbserver-xen is no different than gdb and the standard gdbserver.

2.3.2 Communication between gdbserver-xen and libxenctrl library

The target_ops from the gdbserver-xen such as linux_fetch_registers, linux_store_registers, linux_read_memory, and linux_write_memory use the xc_
Evolution in Kernel Debugging using Hardware Virtualization With Xen

Figure 2: Typical use of gdbserver to remotely debug a Linux process

pttrace interface from the libxenctrl library to exchange the processor state or memory contents for the VMX domain. The xc_ptrace() function implements the ptrace-like mechanism to control and access the guest.

2.3.3 libxenctrl and dom0 privcmd device interactions

Gdb’s request to get and set processor registers is implemented in the libxenctrl library by xc_ptrace() calls like PTRACE_GETREGS, PTRACE_SETREGS, PTRACE_GETFPREGS, PTRACE_SETFPREGS, and PTRACE_SETFPXREGS. Inside the xc_ptrace(), the registers are fetched by the calling fetch_regs() and changed by calling xc_vcpu_setcontext() functions. fetch_regs() uses the dom0_op(DOM0_GETVCPUCONTEXT) Xen hyper call to get the context of the guest vcpu. The hyper call is performed by making IOCTL_PRIVCMD_HYPERCALL ioctl on the /proc/xen/privcmd domain-0 xen device.

For gdb’s request to get or change code (text) and data memory, the xc_ptrace requests like PTRACE_POKETEXT, PTRACE_POKEDATA are used. These ptrace requests use the map_domain_va() function to map the guest memory in the gdbserver-xen process’s address space, and do the read/write operations on that mapped memory. The map_domain_va() function is also part of the libxenctrl library. It finds out the mode the guest vcpu is running in, like real mode or 32-bit protected paging disabled, or paging enabled or paging with PAE enabled, or 64-bit IA32e mode (long mode). If the guest processor is in paging-disabled mode such as real mode then it can get the machine physical address for the address gdb has requested directly using the page_array[] for the domain.¹

Then the xc_map_foreign_range() function is used to map the machine page frame of

¹The page_array[] is an array of the physical page frames numbers allocated to the guest in the guest physical address order. It is built at the time of creating a new domain. page_array[guest_physical_pfn] gives corresponding machine_physical_pfn. The page_array[] for the guest domain is obtained by making DOM0_GETMEMLIST dom0_op() hyper call.
the guest page in the gdbserver-Xen’s address space. It uses \texttt{IOCTL\_PRIVCMD\_MMAP} ioctl on the \texttt{privcmd} domain-0 xen device to map the machine page frames into the the gdbserver-xen process’s address space. Once it is mapped in the gdbserver-xen process’s address space, it then performs simple memory reads or writes to access the guest memory contents.

\texttt{map\_domain\_va()} determines if the guest vcpu is running in paging-enabled mode or not, by looking at control registers received from the \texttt{fetch\_regs()} call for the guest vcpu. The control registers \texttt{CR0} and \texttt{CR4} tells which mode the guest vcpu is running such as real, protected paging disabled, protected paging enabled, PAE, or IA32e mode. The control register \texttt{CR3} points to the guest physical address of the current page table the guest vcpu is using. The IA32 architecture has different-format page tables for different processor paging modes.

The \texttt{libxenctrl} library implements \texttt{map\_domain\_va\_32}, \texttt{map\_domain\_va\_pae}, and \texttt{map\_domain\_va\_64} functions to handle these different page table formats. The \texttt{map\_domain\_va\_64()} handles page tables for both 4k and 2M pages in the IA32e mode.

After getting the guest physical address for the gdb requested virtual address by traversing the guest page tables, then the rest of the functionality to map the guest page frame and perform read/write is implemented similar to the paging-disabled situation described above.

### 2.3.4 privcmd domain-0 device and the Xen hypervisor Interactions

The \texttt{/proc/xen/privcmd} device is implemented as a device driver in the dom0’s paravirtualized kernel. For the device, the \texttt{IOCTL\_PRIVCMD\_MMAP} ioctl is implemented by making a \texttt{HYPervisor\_mmu\_update} hyper-call in Xen. And like all the other hyper calls, the \texttt{IOCTL\_PRIVCMD\_HYPERCALL} ioctl is implemented by making a call at right the offset in the \texttt{hypercall\_page}.

The \texttt{hypervisor\_page} has the handlers for the hyper calls with various parameters. And it is initialized by the Xen hypervisor at the domain creation time. The initialization of the \texttt{hypercall\_page} involves writing the appropriate code in the page for hyper call handlers. For x86\_64 domain-0 the \texttt{hypercall\_page} is initialized with \texttt{syscall} handlers; for i386 it is it is initialized with \texttt{int 0x82} calls. For \texttt{supervisor\_mode\_kernel} i386 domain-0 kernel it is initialized with the \texttt{long call} instruction. All these methods get the processor execution control into the Xen hypervisor, and call the appropriate function from the \texttt{hypercall\_table}. After the execution of the hyper call function, the hypervisor returns control to the next instruction after the hyper call in the dom0, by making an \texttt{iret} or \texttt{long call}, or \texttt{sysret} instruction.

### 2.3.5 The Xen hypervisor infrastructure for debugging

The get and set of vcpu context \texttt{dom0\_ops} described in Section 2.3.3 provide \texttt{gdb} with read/write access to the guest vcpu registers. All the \texttt{dom0\_op} hyper calls are handled by the \texttt{do\_dom0\_op()} function in the Xen hypervisor. The \texttt{DOM0\_GETVCPUCONTEXT} \texttt{dom0\_op} gets the vcpu register context of the guest vcpu by reading the per-vcpu context information for the guest domain stored in the hypervisor. Not all information is available there, because some of the guest register state is stored in VMCS for faster access. It needs to get the register state information from the VMCS to get the complete register state of the guest vcpu.
In a multi-processor system, the domain-0 vcpu running the dom0_op can be different than the guest domain vcpu being debugged. And the VMCS structures are per logical processor; one processor cannot operate on another processor’s VMCS. So if the processors backing these two vcpus are different, then the vcpu running the dom0_op() sends an IPI (InterProcessor Interrupt) to ask the other vcpu to provide its VMCS guest registers’ context state. The VMCS registers’ context state is obtained by temporarily loading the VMCS for the guest vcpu and performing VMREAD instructions for the guest registers.

The DOM0_GETVCPUCONTEXT dom0_op is implemented similarly using the VMWRITE instruction.

The guest memory access is enabled by mapping the guest memory in the gdbserver-xen’s address space using the mmu_update hypercall; this is described in Section 2.3.4. The mmu_update hypercall is implemented in the hypervisor by modifying the page table entries of the gdbserver-xen process running in paravirtualized domain-0, such that the requested virtual address maps to the machine frame number of the requested guest memory location.

The PIT (Programable Interrupt Timer) virtualization for the unmodified guests is also altered in the Xen hypervisor so that it does not try to inject the timer ticks guest has missed due to debugger stopping the guest.

### 2.3.6 Setting breakpoint in the gdb

Gdb sets breakpoints in the guest by placing an int3 instruction at the breakpoint location. Whenever the processor encounters the int3 instruction while executing, a #BP exception is raised, resulting in a VMexit into the hypervisor. The hypervisor handles VMExits based on the exit reason, and for the breakpoint exception VMexit, it simply pauses the guest domain.

Meanwhile the gdbserver is waiting for the guest domain to pause. Once it discovers that it has paused, it uses the xc_ptrace interfaces to get the processor register state of the guest domain’s vcpu, and passes it on to gdb. gdb was waiting for a response from the gdbserver. Once it gets the response, it shows where the guest is paused, by looking at the eip/rip from the guest vcpu register state. At this point, the gdb user can issue various gdb commands to access and manipulate the guest processor and memory state.

When the user asks gdb to continue the debugged guest, gdbserver unpares the guest domain, and waits for it to get into paused state again. The gdbserver also handles the CTRL-C press from its terminal, pausing the domain and returning control to gdb.

### 3 Current Limitations

There are two types of limitations. One is limitation in the use of gdb, because not all the gdb commands are implemented in the gdbserver-xen. And then there are limitations due to the Xen environment of the unmodified guests.

#### 3.1 Limitations on gdb use

Currently breakpoints are implemented only using the int3 instruction and traps. The processor debug registers are not touched. This puts some limitations on the guest debugging.

Currently these gdb capabilities are missing from the gdbserver-xen for debugging an unmodified guest:
• hardware breakpoints
• watchpoints
• single stepping

3.2 Limitations on the guest driver debugging

The unmodified guests running on the Xen sees only those hardware devices emulated by Xen. And as of now, unmodified guests running on Xen can not access the platform devices directly. Hence this debug capability can not be used to debug any arbitrary device driver. Only the device driver for the virtualized devices Xen shows to the unmodified guests can be debugged.

In the future with the Intel Virtualization Technology for Directed I/O [8] and Xen’s capability to assign machine devices directly to the unmodified guests, it will be possible to use this debug capability to also debug the device drivers for the platform’s devices.

4 Comparison with other Linux Kernel debuggers

There are other debugging [9] options available for debugging the Linux Kernel, such as software debuggers: KDB [10], KGDB [11], and hardware JTAG based debuggers: Arium Intarget probe [12].

4.1 KDB

The Linux Kernel Debugger (KDB) is a patch for the Linux kernel and provides a means of examining kernel memory and data structures while the system is operational.

4.1.1 Advantages compared to KDB

• No modification to the Linux kernel is needed. KDB needs a KDB enabled/patched Linux kernel. If the KDB patch for the desired kernel is not available, then there will be more effort to port the KDB patch to the desired Linux kernel.

• Not only Linux, any PC operating system can debugged.

• Can set breakpoint anywhere in the kernel, even in the interrupt handlers. With KDB, the kernel code used by KDB can not be debugged.

• Source-level debugging. KDB does not support source-level debugging.

• Can also debug boot loader or the guest BIOS code.

4.1.2 Disadvantages compared to KDB

• Requires a system with a VT-capable processor.

• With today’s Xen, arbitrary device drivers can not be debugged.

• Single-step is currently not supported. Instead, breakpoints can be used.

• Does not support extra kernel-aware commands. For example, KDB supports ps, btp, and bta commands to show the running processes and their back traces.

4.2 KGDB

KGDB is a remote host Linux kernel debugger through gdb and provides a mechanism to debug the Linux kernel using gdb.
4.2.1 Advantages compared to KGDB

- The Linux kernel can be debugged with just one system. KGDB needs two systems, one running the Linux kernel, and another controlling it.

- Not only Linux, any PC operating system can be debugged.

- No modification to the Linux kernel is needed. KGDB also needs a KGDB enabled/patched Linux kernel. If the KGDB patch for the desired kernel is not available, then there will be more effort to port the KGDB patch to the desired Linux kernel.

- Can also debug boot loader or the guest BIOS code.

4.2.2 Disadvantages compared to KGDB

- Requires a system with a VT-capable processor.

- With today’s Xen, arbitrary device drivers cannot be debugged.

- Single-step is currently not supported. Instead, breakpoints can be used.

5 Debug environment setup

The following sections provide information on what you need and how to set up your own environment for debugging the Linux kernel using Xen and hardware virtualization and use it effectively.

5.1 Requirements

1. Hardware: In order to run modified guest domains on Xen, first you need a system with processor capable of Intel Virtualization Technology. There is a page [13] set up on the xen wiki here for currently released VT-enabled processors; it can help you in finding the right processor. Going forward, all future Intel processors will incorporate Virtualization Technology.

2. Xen VMM: then you need to get a Xen with the gdbserver changes. Any version of Xen after revision 9496: e08dcff87bb2 dated 31 March 2006 should work. Instructions on how to build and install on your Linux box can be found in the Xen user manual [14]. Instead of building, you can take the ready-built 3.0.2 (or newer) rpm from the download section [15] of the Xensource website.

3. gdbserver-xen: this is the remote agent used to attach gdb to a running, unmodified guest. The sources for gdbserver-xen are part of the Xen source code. The tools/debugger/gdb/README file from the Xen source code provides information on how to build gdbserver-xen.

4. gdb: If you are running x86_64 Xen, then you need 64-bit gdb. If you are running x86_32 or i386 Xen, then you need 32-bit gdb. gdb should be provided by your distribution. Xen allows running a 32-bit OS on the 64-bit Xen using VT—in that case, you will need to use the 64-bit versions of gdb and gdbserver.

5.2 Setting up the debug environment

Once you have all the required components, then you can go ahead with the setup as follows.
1. Start the unmodified guest normally on top of xen. If you are not familiar with Xen, you can refer to the Xen user manual [14].

2. get the domain_id for the running guest from the xml list command in the service domain (domain 0).

3. attach gdbserver-xen to the running guest with this command:
   
   ```
   gdbserver-xen localhost:9999
   --attach <domain_id>
   ```

4. Then start gdb in domain 0 or on a remote host. If you have the binary file with symbols for the guest kernel, you can pass it to gdb.
   
   ```
   gdb -s vmlinux
   ```
   You can get this symbol file for various released distributions. Appendix A has more information about it.

   If you do not have such a symbol file for the running guest kernel, you still can debug it by running gdb with no arguments, but you will not be able to use symbols from gdb.

5. Enter the gdb command as shown in Table 1 at the gdb prompt to set up the right environment for gdb. With this, gdb uses the appropriate protocol to communicate with the gdbserver-xen to exchange the architecture state. These initialization gdb commands can also be placed in the .gdbinit configuration file.

6. Enter this at the gdb prompt to attach to the remote gdbserver:
   ```
   target remote <host_running_gdbserver>:9999
   ```

Now you should see the eip/rip where the guest is stopped for debugging. Figure 4 shows the screen shot from starting the gdbserver and gdb connection. It would be more convenient to use separate terminals for the gdbserver-xen and gdb, because you can stop execution of the running guest any time by pressing CTRL-C in the gdbserver-xen terminal.

Now from gdb you can try these commands:

- get registers
  ```
  info all-registers
  info registers
  ```
  ```
  info registers eax
  ```

- set registers
  ```
  set $rip=$rip+2
  ```
  ```
  set $edi=$esi
  ```

- look at memory contents
  ```
  p /4d 0xc00abd23
  ```
  ```
  p page_array
  ```

- change memory contents
  ```
  set *(long *)0xc01231bf=-1
  ```
  ```
  set my_struct[5].my_member=5
  ```
  ```
  set *(long)($rbp +8) = 0
  ```

- look at the disassembly code
  ```
  x /10i $rip
  ```
  ```
  x /10i $eip
  ```
  ```
  x /10i my_function
  ```

- look at the back trace
  ```
  bt
  ```
  ```
  where
  ```
  ```

- set breakpoints
  ```
  break *$rip+0x10
  ```
  ```
  break my_function
  ```
• If you are running on a x86_64 Xen, set the 64-bit architecture in gdb:
  set architecture i386:x86-64:intel

• If you are running on a 32-bit Xen, set the 32-bit architecture in gdb:
  set architecture i386:intel

Table 1: gdb environment setup for gdbserver-xen

• In one terminal start the gdbserver-xen

[root@localhost ~]# xm list
Name     ID  Mem(MiB) VCPUs State Time(s)
Domain-0 0    1024  4  r----- 132.7
ExampleHVMDomain 2    512  1 -b---- 11.7

[root@localhost ~]../gdbserver-xen localhost:9999 --attach 2
Attached; pid = 2
Listening on port 9999

• In another terminal start the gdb

[root@localhost ~]# gdb
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".

(gdb) set architecture i386:x86-64:intel
The target architecture is assumed to be i386:x86-64:intel

(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
0x00000000c0109093 in ?? ()
(gdb)

Figure 4: starting the gdbserver-xen and gdb connection
• continue to get the breakpoints hit
  cont

• delete breakpoints
  delete 1

• advance to some location
  advance my_function

• jump execution to some other location
  jump *0xffffffff8010940f
  jump crash.c:90
  jump my_function

6 Examples

• Figure 5 shows how to get the debuginfo for the stock Red Hat Enterprise Linux 3 Update 5 kernel, and use it with gdb to debug the kernel at source level.

• Figure 6 shows a custom-compiled x86_64 2.6.16 Linux kernel debugged at source level on 64-bit Xen.

• Figure 7 shows a freedos beta9rc5 debugging without the symbols information.

7 Summary and Conclusion

The paper describes unmodified Linux kernel debugging at source level using Xen on platforms with hardware virtualization processors. It describes how the gdb commands gets implemented in the various components of Xen. It shows how to set up this debug environment and provides a high-level comparison between other Linux kernel debuggers.

Appendix A

Debug info for stock distribution kernels. Figure 5 show how to use this kernel-debuginfo rpm to debug the Linux kernel at source level.

• Red Hat Fedora Core Distributions:
  http://download.fedora.redhat.com/pub/fedora/linux/core/{1,2,3,4,5}/i386, x86_64}/debug/

• Red Hat Enterprise Linux Distributions:
  http://updates.redhat.com/enterprise/{3AS, 3ES, 3WS, 3desktop, 4AS, 4ES, 4WS, 4Desktop}/en/os/Debuginfo/{i386, x86_64}/RPMS/


• Other Linux Distributions: I could not find the debuginfo rpms for other distributions.

Acknowledgments

Thanks to my colleagues at Intel Open Source Technology Center and friends in the open source community for their continuous support.

Thanks to Keir Fraser and Ian Pratt for providing me comments and suggestions for my gdbserver-related patches for unmodified (VMX or HVM) guests to Xen.

Thanks to John W. Lockhart for helping me format this paper in \LaTeX.
Figure 5: An Example of source-level debugging of a 32-bit Red Hat RHEL3 Update 5 stock Linux kernel on 64-bit Xen.
Figure 6: Example of source-level debugging of a 64-bit custom-compiled 2.6.16 Linux kernel on 64-bit Xen.
Figure 7: Example of debugging freedos beta9rc5 on 32-bit Xen.
References

http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.


http://www.gnu.org/software/grub/.


[8] Intel virtualization technology for directed i/o architecture.  

[9] Steve Best. Linux debugging techniques article.  
Technical report, IBM Linux Technology Center.  

http://oss.sgi.com/projects/kdb/


http://wiki.xensource.com/xenwiki/IntelVT.

http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/user/user.html.


Disclaimer

The opinions expressed in this paper are those of the author and do not necessarily represent the position of the Intel Corporation.

Linux is a registered trademark of Linus Torvalds.

Intel is a registered trademark of Intel Corporation.

All other trademarks mentioned herein are the property of their respective owners.
Evolution in Kernel Debugging using Hardware Virtualization With Xen
Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada
Conference Organizers

Andrew J. Hutton, Steamballoon, Inc.
C. Craig Ross, Linux Symposium

Review Committee

Jeff Garzik, Red Hat Software
Gerrit Huizenga, IBM
Dave Jones, Red Hat Software
Ben LaHaise, Intel Corporation
Matt Mackall, Selenic Consulting
Patrick Mochel, Intel Corporation
C. Craig Ross, Linux Symposium
Andrew Hutton, Steamballoon, Inc.

Proceedings Formatting Team

John W. Lockhart, Red Hat, Inc.
David M. Fellows, Fellows and Carr, Inc.
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.