Loadable Hypervisor Modules

Thomas Naughton, Geoffroy Vallée, and Stephen L. Scott∗
Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831, USA
{naughtont, valleegr, scottsl}@ornl.gov

Ferrol Aderholdt†
Tennessee Technological University
Department of Computer Science
 Cookeville, TN 38501, USA
wfaderhold21@tntech.edu

Abstract
This paper discusses the implementation of a new hypervisor mechanism for loading dynamic shared objects (modules) at runtime. These loadable hypervisor modules (LHM) are modeled after the loadable kernel modules used in Linux. We detail the current LHM implementation based on the Xen hypervisor. Potential use cases for this LHM mechanism include dynamic hypervisor instrumentation for debug tracing or performance analysis. We discuss the initial LHM prototype and future plans.

1 Introduction
Currently, virtualization is being used for a variety of purposes. The ability of system-level virtual machines (VMs) to decouple the operating system from the hardware has spurred their use in the area of server consolidation to improve system utilization. This can help to reduce the number of physical machines and reduce the associated operation costs, e.g., power, cooling, etc. The encapsulation of VMs can be leveraged to assist in system management and provide user customizable environments [5, 14, 17]. Virtualization also provides interesting opportunities for fault tolerance, e.g., VM migration for proactive fault tolerance [10]. These capabilities are being leveraged for research and development in a range of domains, to include High-Performance Computing (HPC) [18].

Regardless of the end use, the ability to keep the virtual machines up and running is critically tied to the virtual machine monitor (VMM), or hypervisor. The shutdown or restart of a hypervisor forces all state associated with the active VMs to be saved (or migrated). Therefore, it goes without saying that hypervisor restarts should be limited as much as possible. Additionally, the hypervisor capabilities should be limited to those needed for the “jobs” run at that site, i.e., avoid VMM features that are not relevant to site’s uses.

As virtualization gains adoption, be it for large-scale HPC systems or small embedded environments, the memory consumption of the hypervisor should be minimized. Therefore, it can be useful to limit the base set of functionality and dynamically load additional capabilities, as needed, at runtime. The key motivation for this work is to provide a mechanism for such dynamic modification at the hypervisor level. This mechanism can be used in providing more advanced features like runtime customization. It is also useful for supporting dynamic debugging and tracing services.

The remainder of this paper is structured as follows: Section 2 highlights some related background material; Section 3 discusses the design and implementation details; and Section 5 and Section 6 cover future work related to the use of LHMs and concluding remarks.
2 Background

This section provides basic background and terminology used throughout the remainder of the paper.

2.1 Virtualization

There are many approaches to virtualizing resources, which can be implemented at various levels of the software stack. The focus of this work relates to system-level virtualization, where an additional layer is interposed between the hardware and software interface (layers). The following discusses the components and terminology associated with this virtualization layer.

A virtual machine (VM) is an execution environment that runs atop a layer of software, which provides a virtualized view of the physical resources. A virtual machine monitor (VMM), or hypervisor, is responsible for the management of these VMs. Three basic criteria were outlined by Popek and Goldberg [12] that provide the general requirements for a VMM: (i) fidelity – VMM environment is “essentially identical” to the physical machine, (ii) efficiency – VMM runs most operations on physical resources, (iii) control – VMM fully controls the physical resources. Depending on where the VMM resides in the software stack it may fall into one of two broad classes [6]: type-I or type-II. The basic distinction being where the VMM operates with respect to the physical hardware (see Figure 1). In a type-I VMM the monitor sits directly on the hardware, e.g., Xen. The type-II VMM resides above a host operating system (i.e., as a process). Both VMM types are responsible for the management and proper control of the guest virtual machines.

Figure 1: Goldberg virtualization classifications.

The operating systems (OSs) that run on the physical host and in the virtual machine are referred to as the hostOS and guestOS, respectively. In some instances the virtual machine can run an unmodified guestOS, possibly using hardware enhancements like Intel-VT or AMD-V. This unmodified guestOS version is called full-virtualization. However, in some cases like para-virtualization [1, 20] the guestOS is modified, often to improve performance.

2.1.1 Xen

Xen is an open-source hypervisor that implements a type-I virtual machine monitor [1]. The Xen hypervisor uses the term domain when speaking about the operating systems running on top of the VMM. There are two general types of Xen domains: User (domU), and Administrative (dom0). There is only one dom0, and it provides services to the VMM like device drivers, and an end-user interface, e.g., login shells, etc. The calls made from the domains to the hypervisor are referred to as hypercalls. These hypercalls are analogous to system calls for typical operating system, i.e., calls from user-space to kernel-space.

2.2 Loadable Modules

There are several approaches to providing dynamic modification to systems at runtime. Loadable kernel modules are a common technique for providing runtime customization at the operating system level, i.e., they allow new kernel-level code to be loaded at runtime. In [4], they classify the approaches to system-level modifications based upon: (i) initiator of the adaptation (e.g., human, application, OS), and (ii) time of the adaptation (e.g., static: design, build, install, or dynamic: boot, runtime). For example, based upon this classification the Linux Loadable Kernel Modules (LKM) provide dynamic human initiated customization of the OS kernel.

The following section will provide a high-level overview of the Linux modules subsystem. The emphasis is on aspects that are pertinent to the work done for loadable hypervisor modules.

2.2.1 Linux: Loadable Kernel Modules

The Executable and Linkable Format (ELF) is a standard for creating and interacting with binary object files [16].
A Linux Loadable Kernel Module (LKM) is a relocatable ELF (ET_REL) file [16]. This format allows the symbol addresses to be adjusted to the final locations used at load time, i.e., relocated. The Linux build system inserts additional sections to the ELF file, which may later be used by the modules subsystem. For example, if kernel version checking is enabled, the module will include a section with checksums (versions) for symbols that are used as a sanity check at module load time. The logical structure of a module file is shown in Figure 2.

The modules are loaded into the running system via a system call, e.g., `init_module()`, which is typically called by user-space utilities like `insmod` or `modprobe` [2, 8]. The Linux v2.6 kernel contains an “in-kernel” loader that does the majority of the work involved in resolving references and loading the module files into memory. This greatly simplifies what is done by the user-space utilities, and helps with maintenance of the LKM subsystem [7, 8, 13]. As noted above, the module files contain additional sections that are used to convey information to this in-kernel loader. This approach can be helpful when making changes to the modules subsystem, as will be discussed in later sections, since all functionality is maintained in the kernel (and to some extent the build system).

The kernel can also be configured to provide information about the location of symbols in memory. When compiled with this option, listing the contents of the pseudo-file, `/proc/kallsyms`, displays the address, type, and symbol names for the kernel. If the symbol is from a loaded module, the module’s name is also shown. When a module is loaded (or unloaded) this information is reflected in the output of `/proc/kallsyms`. Note, when a module contains an undefined symbol (i.e., symbol type is ‘U’) the output will show the address for the actual symbol’s address. For example, external routines defined outside the module like `printk()` would show the address for the actual definition in the kernel. The in-kernel loader is responsible for making these sorts of changes during module load.

Once the module has been loaded into memory and prepared for execution, various system entries are created for the newly loaded module, e.g., add entries to `/sys/module` and `/proc/module`. Then, if one exists, the module’s initialization routine is called, and the
state of the module is marked as “ready”. The basic steps involved in loading a LKM are outlined in Figure 3.

3 Loadable Hypervisor Modules

The general motivation and initial idea of providing a dynamic adaptation facility was discussed in [11]. The high-level design goals for the new mechanism were to keep the interface simple and wherever possible reuse existing code to speed development. After reviewing the Linux LKM implementation, we decided to follow a similar approach by extending the Xen hypervisor to support loading new code at runtime, i.e., Loadable Hypervisor Modules. The fact that Xen reuses portions of the Linux code base, and shares a common object file format (ELF) helped to satisfy our goal of reusing existing code where possible. Also as discussed in Section 2, the fact that Linux has an in-kernel loader enabled us to make changes to the Xen hostOS and keep the user-level tools (interface) relatively unchanged*. As with standard Linux modules, only the administrator can load or unload an LHM. Therefore the security of LHMs is managed in the same manner as other modules used by the administrative domain (dom0).

The current implementation reuses the standard Linux build environment when compiling LHM files. There is an additional section added to the ELF binaries that denotes the module is to be loaded into the hypervisor. The hostOS version of Linux was enhanced to recognize this special section, .lhm.dummy_hello (Figure 2).

In Xen, the space for dynamic memory (heap) is rather small. To avoid arbitrarily increasing the size of Xen’s heap we instead “map” areas of the hostOS memory into the hypervisor and only use additional memory resources as needed when loading LHMs. Upon module load, the hostOS will allocate space in the kernel and copy the contents of the ELF binary from user-space. The module is prepared for execution as normal, with the exception that undefined symbols are resolved using addresses from the hypervisor. Therefore, the module will reference module-only symbols, and/or externally defined symbols in the hypervisor.

The layout of symbols in the hypervisor is generally static since there are no runtime changes. A new mechanism to access information about the symbols in the hypervisor was added, i.e., information like /proc/kallsyms under Linux. This is useful for showing the location of static information in the hypervisor as well as dynamically loaded code. This information is also used during module load to lookup undefined symbols in the LHM and patch the reference with the appropriate address. The information is available through a new hypervisor call, do_hallsyms, and is exported to the hostOS via /proc/hallsyms.

The enhanced hostOS kernel does the loading and patching of undefined symbols for the LHM. Once the module is ready for insertion, a hypercall is made that maps the module into the hypervisor. Since the amount of heap space is limited in the hypervisor, the module remains in the memory region allocated by the hostOS. The hypervisor has access to all the memory so this is acceptable, and the regions are write-protected from the hostOS to avoid any accidental accesses. When the LHM is unloaded, the memory resources are released for reuse by the hostOS.

A logical layout of the mapping for a LHM is shown in Figure 4. In the figure, the memory regions are indicated by the starting four hex digits:

ff11 xxxx – LHM address in Xen range
cfad 2020 – LHM address in Linux range
cfad 1010 – General Linux address

The steps involved in loading an LHM are summarized in Figure 5.

3.1 Prototype Status

The current prototype was developed and tested using Xen-3.2.1 and the associated hostOS Linux-2.6.18-xen from the Xen Mercurial repository. The supported architectures are x86-32 and x86-64†.

We added two new hypercalls to Xen as part of this initial implementation. The first, do_lhm_op() is used for LHM loading (“mapping”) and unloading. The other hypercall is used for gathering hypervisor symbol information, do_hallsyms().

---

*Note, minor changes were made to rmmod to work around hard coded values like /proc/modules.

†On x86-64, the GCC option --mmodel=large must be used in order to generate 64-bit relocation types in LHM files.
4 Evaluation

The basic LHM mechanism has been developed and tested on both x86-32 and x86-64 platforms. In Section 5 we discuss an initial use-case for this mechanism, but that work is not yet complete. Therefore, our evaluation is based on the source code itself, which can give insights into the size and complexity that was added to the hypervisor.

Note, these numbers are based on a development version of the source tree, i.e., prior to any major code cleanups for the initial release. They provide a general idea of the location and rough size of the modifications. The numbers were gathered using the sloccount utility by David A. Wheeler, which gathers the “physical source lines of code (SLOC)” [19]. This utility is able to properly count SLOC for a range of different languages — ignoring comments, whitespace, etc.

The current source tree has 874 SLOC added for Linux (dom0), and 1953 SLOC added for Xen (vmm). These numbers include both the basic LHM functionality and the current implementation of the instrumentation mechanism discussed in Section 5. Therefore, if you remove the instrumentation related files, these numbers are closer to ≈785 SLOC and ≈560 SLOC, respectively. Note, the hypervisor symbol (hallsyms) export code accounts for ≈305 SLOC and ≈178 SLOC of these total non-instrumentation numbers.

In summary, the Linux (dom0) portion of the LHM code is roughly ≈480 SLOC with an additional ≈305 SLOC for accessing hypervisor symbols. There are ≈382 SLOC for Xen (vmm) plus ≈178 SLOC for exporting hypervisor symbols. The following tables break those values down to show numbers based on the region of the source tree i.e., sub-directory (Table 1), and particular (significant) source files (Table 2).

In Table 1 a summary of the line count is provided. These counts include both the Linux (dom0) and Xen (vmm) code and reflect the major files that were updated or created. The numbers omit the small set of additions to support the three new system calls under Linux and three new hypercalls under Xen. Also, there was a minor work around added for an issue with Xen console output from LHM code.

5 Future Work

Our next steps are to continue development and testing of the LHM mechanism. We are also in the process of looking at future use cases for the new mechanism. The first
Table 1: Source Lines of Code (SLOC) for selected files. Note, the “updated” files reflect the new lines added (e.g., +150).

<table>
<thead>
<tr>
<th>Location</th>
<th>Filename</th>
<th>SLOC</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (dom0)</td>
<td>module.c</td>
<td>+150</td>
<td>updated</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>module.h</td>
<td>+91</td>
<td>updated</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>hallsyms.c</td>
<td>301</td>
<td>created</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>hallsyms.h</td>
<td>4</td>
<td>created</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>module_fix.h</td>
<td>75</td>
<td>created</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>public/lhm.h</td>
<td>57</td>
<td>created</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>instrument.c</td>
<td>69</td>
<td>created</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>instrument.h</td>
<td>18</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>hallsyms.c</td>
<td>154</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>hallsyms.h</td>
<td>8</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>lhm.c</td>
<td>185</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>public/lhm.h</td>
<td>4</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>xen/lhm.h</td>
<td>96</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>module.h</td>
<td>+50</td>
<td>updated</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>instrument.c</td>
<td>583</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>instrument.h</td>
<td>25</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>intel_op.c</td>
<td>55</td>
<td>created</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>intel_op.h</td>
<td>731</td>
<td>created</td>
</tr>
</tbody>
</table>

Table 2: Source Lines of Code (SLOC) by directory with source file type.

<table>
<thead>
<tr>
<th>Location</th>
<th>Source Dir</th>
<th>File Type</th>
<th>+SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (dom0)</td>
<td>arch/</td>
<td>ASM</td>
<td>+1</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>include/</td>
<td>ANSI C</td>
<td>+342</td>
</tr>
<tr>
<td>Linux (dom0)</td>
<td>kernel/</td>
<td>ANSI C</td>
<td>+531</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>xen/</td>
<td>ANSI C</td>
<td>+1941</td>
</tr>
<tr>
<td>Xen (vmm)</td>
<td>xen/</td>
<td>ASM</td>
<td>+12</td>
</tr>
</tbody>
</table>

use case that is currently under development is the creation of a hypervisor-level dynamic instrumentation facility. This work was motivated by prior research into system level instrumentation, e.g., KernInst [15], DKM [9] and DTrace [3]. This new hypervisor mechanism will allow probes to be loaded at runtime, which can be used to add additional code to existing functions or provide a full replacement for the given function. These Hypervisor Instrumentation Points (HIPT’s) are limited to the following four locations:

- Type-1: code replacement of an entire function,
- Type-2: code splicing at the function entry,
- Type-3: code splicing at the function exit, and
- Type-4: code splicing at pre-defined locations within the source code specified by the user.

Here the term “code splicing” means inserting new functionality to the code, and “code replacement” means wholesale replacing functionality in code by overwriting a portion [15].

The current goal is to allow for insertion of a HIPT without modification to the target source code, with one exception. In a Type-4 HIPT, the source code would be modified to insert null locations where new code could be spliced at runtime. Note, we currently plan to add the restriction that a HIPT will only be applicable to subsequent invocations of the routine, i.e., no attempt is made to patch addresses already in use on the stack (non-quiescent functions).

6 Conclusion

As virtualization becomes more prevalent, tools to aid in debugging and performance analysis become more important. Often it is undesirable to keep these tracing or debug routines enabled at all times due to the added overhead they may introduce. In prior research, dynamic customization has been used to extend system-level capabilities at runtime, e.g., Linux’s loadable kernel modules.

In this paper we have described a new mechanism that supports dynamic loading of modules (code) at the hypervisor level. This functionality allows for modifications to the hypervisor at runtime without having to shutdown...
or reboot the system. These Loadable Hypervisor Modules (LHM) are modeled after the loadable kernel modules provided by Linux. The implementation was done using Xen 3.2.1 and supports loading and unloading of modules on x86-32 and x86-64 platforms.

This new mechanism can be beneficial in providing more advanced features like dynamic tracing. We briefly described an initial use case for LHM's that is currently under development and provides a basic hypervisor-level instrumentation facility. As part of future work, we plan to complete the support for hypervisor instrumentation points (HIPT’s) and explore additional use-cases for LHM’s.

References


