First Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007)

Dynamic Adaptation using Xen *

Thoughts and Ideas on Loadable Hypervisor Modules

Thomas Naughton Geoffroy Vallée Stephen L. Scott

Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831, USA
{naughtont, valleegr, scottsl}@ornl.gov

Abstract

The topic of virtualization has received renewed attention. Xen is a popular open source type-I hypervisor. The Xen hypervisor currently has limited capabilities for runtime modification to the core hypervisor, which impairs research into dynamic adaptation for system-level virtualization. This paper discusses recent investigations into the feasibility of extending Xen to support runtime adaptation for core hypervisor service, e.g., scheduler.

Categories and Subject Descriptors D.4 [*Software*]: Operating Systems

General Terms Virtualization, Adaptation

Keywords Xen, Linux, Loadable modules

1. Introduction

The recent resurgence in virtualization has led to new avenues of research leveraging virtual machine monitors (VMM), also referred to as *hypervisors*. These hypervisors provide the basic services needed to manage virtual machines (VMs) and in the case of Xen leverage existing host operating systems (host OS) for device drivers, etc.

The Xen hypervisor [1] has gained much attention in recent years. This interest is in part due to the fact that it is open source and provides an avenue for research and experimentation into system-level virtualization. However, Xen provides no facilities for dynamic (run-time) adaptation of the core hypervisor. That is to say, any changes to the hypervisor require a recompilation and reboot, which requires any running VMs to be stopped and saved (check pointed).

The ability to perform dynamic adaptation of a running hypervisor can be helpful for a number of reasons. Examples include, adding new mechanism/capabilities or simply providing alternatives to existing services and policies, e.g., the VM scheduler. These dynamic hypervisor modules also can be beneficial during development phases where new approaches are prototyped and/or profiled, or simply for upgrading portions of the system. Such an adaptation mechanism has been implemented in the Linux kernel, via the dynamic management of kernel modules. This mechanism may be used as a reference for the implementation of mechanisms for dynamic hypervisor adaptation.

The remainder of this paper focuses on the aspects related to adding such a feature to Xen v3. We begin with some general background material in Section 2 followed by a brief discussion regarding the motivation for such a feature in Section 3. Then we outline some of the basic steps involved and discuss some details based on Linux's loadable kernel modules in Section 4. In Section 5 we mention some related work. Lastly, in Section 6 we mention future work and provide concluding remarks.

2. Background

In a later section we discuss Linux's loadable kernel modules. These modules are roughly analogous to the loading of dynamic libraries in user-space applications. This section will review a few of the details related to object files and associated tools, which come in to play when providing runtime alterations, i.e., system-level dynamic modules.

The Executable and Linkable Format (ELF) is a standard for creating and interacting with object files in a portable fashion [13]. The ELF file format is commonly used in UNIX like systems – Linux uses ELF as does the Xen hypervisor. The object file encodes the data and code for a program, i.e., compilation of a high level language to a lower level machine oriented format. The file is structured into sections or segments that are used by the linker (link editor) or loader (dynamic loader), respectively.

There are three types of ELF object files [7, 13]: (i) *relocatable*, (ii) *shared*, and (iii) *executable*. The first two are involved with program linking to different extents. They differ in that *relocatable object files* provide data and code (symbols) that can be used to create shared objects or executables

* This work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

HPCVirt'07 March 20, 2007, Lisbon, Portugal.

but *shared object files* are also shared libraries that can provide code for direct runtime execution. The *shared object files* are used to dynamically load commonly used routines at program load time (dynamic loader). Lastly, *executable object files* are just as the name implies, those files that are suitable for stand alone execution, i.e., all symbols have been resolved, barring any shared objects that are loaded at runtime [7, 13]. An example for the three ELF object types, taken from Linux v2.6 system files: (i) relocatable – device driver module `e100.ko`[1], (ii) shared – standard C library `libc.so.6`, and (iii) executable – uncompressed kernel `vmlinux`.

As mentioned above the ELF files are used by the linker and loader. The role of these two tools is quite similar and basically involves the replacement of generic references to the actual definitions (linker) and then the loading of the data into memory for execution, which might involve relocating reference as well (loader). In Figure 1, the basic layout is shown for an executable ELF object file, e.g., Linux kernel *vmlinux*. These sections (or segments when working with loading) organize the various aspects of the object file and ultimately are used to setup the process image upon execution (see [13] for further details). The linker is typically the final phase of compilation and is called `ld`. The loader is used to resolve references[2], e.g., to shared libraries, whenever the program is being loaded into memory for execution. The loader is typically called `ld.so` or `ld-linux.so` under Linux.

ELF Header
Magic: 75 45 4c 45 ...
...
Type: EXEC

Program header table
LOAD 0x001000
...

Section NULL

...

Section __ksymtab

...

Section .symtab

Section header table
[0] NULL ...
...
[5] __ksymtab PROGBITS ...
...
[30] .symtab SYMTAB ...

Figure 1. ELF file layout with example content similar that found in a Linux kernel image (based on [13] Figure 1-1)

3. Motivation

The operating system provides services to applications and manages available resources. What services and how the resources are managed are key design decisions that influence program execution. The ability to tailor an execution environment to suite application needs can be done at many levels but often these are fixed at compile time. One approach to providing a customized environments is to support adaptation at the lowest levels. In the context of virtualized environments this adaptation could take place at either the virtual machine (guest OS) level or at the hypervisor level.

There are several approaches for achieving customization and adaptation of operating systems and runtime environments [5]. The two main criteria identified by [5] for classifying these approaches to system-level customization are: (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).

Based on this taxonomy, the Xen hypervisor supports static human initiated customization, i.e., change the code manually, build, and boot. This is an acceptable level in many cases but disallows the ability to do dynamic adaptations at runtime, regardless of how they are initiated. Therefore, in order to do experimentation of core hypervisor services and policies at runtime some level of dynamic adaptation is necessary. Otherwise, the entire state of the system, running Virtual Machines, etc., must be stopped in order to make even minor changes to the hypervisor. Additionally, this capability would allow for basic runtime patching to the hypervisor.

As observed by [12], in some instances system-level adaptation may target improved execution time for an application, where in other circumstances it may be more appropriate to focus on things like fairness or responsiveness. Having the capability to perform dynamic adaptation is the first step toward more ambitious goals such as automatic adaptation at runtime [5, 12].

This capability also provides a basic mechanism to enable customization for things like Quality of Service (QoS) or Fault Tolerance (FT) policies without having to tear down the execution environment. This could be beneficial for changing workloads or even different phases within a particular application set. These capabilities may ultimately be requested by the virtual machines which house the applications, but lower level services might need to change to support such features, e.g., I/O schedulers or VMM by-pass facilities.

[1] The kernel modules show as relocatable (REL) via `readelf` but details from the Linux Kernel Mailing List (LKML) and other references [10] lead us to believe they are a special form of a relocatable ELF binary with some traits of shared binaries (ET_DYN).

[2] Due to this linkage editing during the image load, the loader is sometime referred to as the “dynamic linker” or “dynamic loader”.

Another motivation for runtime system-level changes is to support profiling or debugging. The use of fine-grained techniques like those employed by "KernInst" [11] allow for changes to the machine code that splice (insert) jumps to patches that can be used for dynamic instrumentation. This can then be used to target specific functions for performance reasons, diagnostic purposes, or possibly to specialize for a given application (process).

Lastly, an argument could be made to simply provide alternate capabilities into the hypervisor at compile time and simply select from the available options at runtime. However, this limits the extent to which changes can be made and eliminates the ability to do dynamic experimentation without having to stop-rebuild-boot for each variation. This would also increase the size of the hypervisor footprint and consume more resources for potentially useless services. This is unacceptable, especially in an HPC context, because you need to keep the size of the hypervisor as small as possible to limit the memory overhead.

A common example of system-level dynamic adaptation is the loadable kernel modules found in systems like Linux [5, 6]. In the next section we summarize the approach taken by Linux, using it as a potential basis for adding such functionality to Xen.

4. Dynamic Hypervisor Adaptation

Recently we began looking into the feasibility of adding dynamic adaptation to Xen. In order to gather a better grasp of the techniques required to implement such functionality we began with an investigation into the Linux modules facility. Xen and Linux share a common binary format (ELF), and Linux modules are simply relocatable ELF object files [2, 8]. This is an important factor because it forms the basis for the executable memory image. Another reason we chose to look at Linux is that it has supported some form of dynamically loadable modules since Linux v1.0/1.2 [6, 9]. Also, starting with v2.6 the entire modules system has been reimplemented with most of the work being done directly in the kernel [4, 6, 8].

4.1 Overview of the Linux Module Mechanism

The loading of a module begins with the user-space application `insmod`, which attempts to install a given module into the running kernel. There are more intelligent tools like `modprobe` that are typically used to resolve module dependencies but ultimately they use the same mechanisms to load the module. Once the module file has been read the system call `init_module` is invoked, which attempts to load the module into the running kernel. A rough outline of the steps involved follow [8]:

1. allocate user-space buffer and fill with contents of relocatable ELF object (module file)
2. call `init_module()`
3. enter kernel space, `sys_init_module()`
4. check permissions & do needed locking
5. call `load_module()` to allocate space and copy data from user-space to kernel-space. Setup symbols, relocate references, exception table, update instruction cache, etc.
6. link module into list of available modules
7. free lock and notify system of newly loaded module
8. initialize newly loaded module
9. acquire lock and update module state (fully loaded)
10. remove any temporal storage and finalize module book keeping
11. free lock and return

The general approach is similar to the process of the standard linker and loading done at user space. However, in this case the kernel module files are prepared in user space using the standard GNU tool-chain. The build system creates stub files that are combined with the module object files to embedded additional data, e.g., module version information, into the resulting binary kernel modules, i.e., `*.ko` [8, 10]. This approach helps to streamline the in-kernel dynamic loader. For example, the exported symbols for the module are added to a program defined section ("PROGBITS"), e.g., `__ksymtab`, and/or `__ksymtab_gpl` [6, 8, 13].

There are hooks to tie the kernel symbol table to a `/proc` entry that allow for simple access to the current address/symbol mappings. This shows both the static kernel information as well as all dynamic additions.

The approach taken by Linux to provide dynamic adaptation is relatively straight forward, although the implementation is far from trivial, and is widely used. It enables a privileged user to add or remove data/code to a running operating system. Based on these ideas, it seems that a similar approach could be used in the Xen hypervisor. They share the same object file format and some of the Linux implementation might be transferable to Xen. At this stage it seems interesting and would provide a nice enhancement for Xen adaptation efforts.

This capability would allow for experimentation with alternate policies at the hypervisor level, which could be adapted on the fly. For example, these mechanisms could be used to investigate alternate scheduler policies for the virtual machines.

Our investigations are just beginning but the current plans are to begin to add basic support facilities, e.g., symbol table and related routines for exporting symbols, etc. Then experiment with switching between modules that are loaded at boot time, to ensure proper operation before introducing dynamic loading/linking. In parallel, we plan to investigate how dynamic adaptation can be leveraged by the hypervisor and to identify relevant candidates. Initially we plan to target

the Xen scheduler so that we can make changes at runtime to the scheduling of virtual machines.

4.2 Loadable Hypervisor Modules

Here we will discuss some of the considerations and issues involved with adding dynamic adaptation to the Xen hypervisor in the form of runtime loadable hypervisor modules. To help structure the discussion we will consider the example of using such a facility to perform runtime adaptation to the Xen scheduler.

Xen Terminology A few comments regarding Xen terminology. In Xen, *domains* house the *guest virtual machines*. The first domain, *domain0*, is a special case that provides the control interface to the hypervisor and is often referred to as the *host OS*. Xen (hypervisor) and domain0 (host OS) interact through published interfaces called *hypercalls*, which are analogous to system calls in a traditional kernel. The hypervisor is responsible for allocating and then marshalling the physical resources. For example, scheduling of multiple domains (virtual machines) for access to the CPU and for coordinating any privileged instructions they may perform. This latter aspect is essential in order to maintain proper isolation between domains.

Hypervisor Symbols The addresses for in-hypervisor code and data, i.e., hypervisor symbol table, will need to be accessible in order to do the dynamic loading of modules. The modules will need access to existing data and symbols in order to access global variables and common code routines, e.g., global scheduler queues, `spin_lock_irq()`. The modules may also want to export data for use by others. This can then be used to resolve references in the binary modules.

It might be beneficial to make such data available from the host OS via a standard `/proc` interface, by adding an additional hypercall to access the hypervisor symbol table. As mentioned previously, in Linux v2.6 references are resolved via an in-kernel dynamic loader. A similar approach could be taken with Xen by providing an in-hypervisor dynamic loader, or you could use information from the `/proc` area to do the relocation at the host OS (domain0) level.

Allocation and Loading The loading of the module will span several steps. The initial step being the loading of the ELF binary module into memory. This could be achieved in a number of ways but it will originate from one of the domains, likely domain0 (host OS). A transfer would then take place to copy (or map) this into the hypervisor space, which would dynamically allocate space, e.g., `xmalloc()`. As mentioned above, the relocation could be done in the host OS or possibly through an in-hypervisor dynamic loader.

The dynamic modification of the hypervisor will require the addition of some basic mechanisms to manage "modules". This will include some sort of bookkeeping to track the name and location of the module. A hypervisor symbol table will be used in this process in order to locate any common data/code used by the dynamic modules. Additionally,

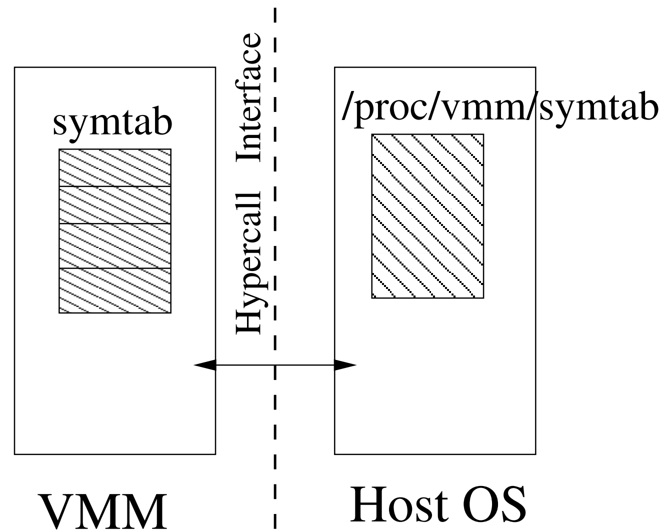

Figure 2. Hypervisor symbol table access from the Host OS.

any dynamic additions that are intended to replace existing hypervisor resident aspects will need access to the symbol table in order to locate and ultimately redirect calls to the newly loaded replacement instance. For example, if you are planning to replace the scheduler with an alternate implementation you will need to update the current address for the scheduler with the alternate version, i.e., splice in the location of the alternate scheduler.

State Management The loading of modules into a running hypervisor will require state management for the individual modules, i.e., loading, ready, unloading. The procedure will transition from the loading and allocation phase into some module specific initialization phase. Once the module is ready for use it would move to ready and be useable by the rest of the system.

As mentioned in Section 5, a key differentiating factor among approaches to dynamic adaptation is the extent to which they require quiescence of state. Teller et al. [12] note that the management of state-full and state-less services is a key challenge when working with dynamic adaptation. Therefore, the degree of isolation for a given service influences the extent to which you can dynamically adapt.

Returning to our modular Xen scheduler example, the API for the routine would be fixed and the domain queues would be global data. Once the new module was available (ready) the global symbol table would be updated to reference the address of the alternate scheduler. This switch would require quiescence of the domains to some acceptable state, and assumes the hypervisor supports some facility to acquire a global mutex to perform the change and then resume execution. It is unclear at this point how things like VMM-bypass would effect matters. However, it is clear that the ordering of events will be important as the hypervisor will use domain0 during module load, and then when chang-

ing to the new scheduler all domains will be paused during the switch to the alternate scheduler. Lastly, the code splicing work by Tamches [11] could be another approach to avoid some of the locking during the switch.

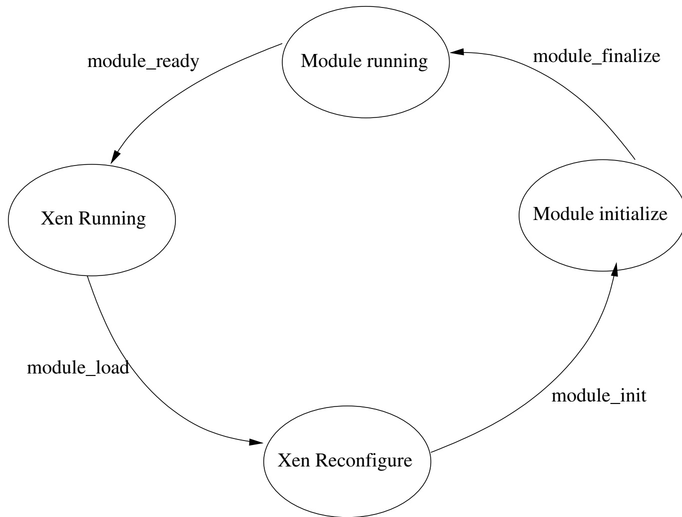

Figure 3. Xen state transitions for loading a module.

5. Related Work

The topic of dynamic operating system adaptation has been explored by a number of projects, with a good survey provided by Denys et al. [5].

As mentioned previously, Linux offers dynamic kernel modules, which provide a mechanism to make changes to select portions of kernel code and data at run-time. This is most commonly used in Linux to provide support for device drivers at run-time, which eliminates the need to compile all possible drivers into one monolithic kernel image. It is worth noting that the Linux module system underwent a major change between Linux v2.4 and v2.6. The majority of the work is now done directly by the kernel with only a minimum of work being performed by the user space tools [2, 4, 6, 8].

The work by Teller and Seelam [12] mentions that the problem of policy adaptation is complex, noting for example the issues of dealing with state-full and state-less resources. Their work has focused on I/O scheduler policy adaptation with experimentations done using the Linux kernel. The work provides some guidelines for dynamic operating system adaptation in HPC environments.

The "KernInst" tool by Tamches and Miller [11] enables dynamic instrumentation of unmodified kernels. Their work was done using Solaris but the techniques should be applicable to other systems. KernInst provides mechanisms for splicing (inserting) alternate code at runtime into a commodity operating system. Their approach allows for fine-grained changes that can be used for dynamic adaptation or for performance analysis purposes.

In [3], Chen et al. have a slightly different objective than hypervisor level modifications, namely live updates to Linux kernels running in virtual machines. Their work is similar in nature to that of Tamches et al. but their use of system-level virtualization enables a slightly different approach which uses binary patches to splice (insert) changes into the running VM and perform patching without having to stop or even quiesce the system (VM). The hypervisor also controls the VM's page tables in order to instrument traps when making changes to existing code/data. Although they focus on VM changes at runtime and not the actual hypervisor, the motivations are similar and some of their techniques might be of interest, e.g., "stack inspection" and hardware trace facilities (execution stepping).

6. Conclusion

The ability to perform system-level dynamic adaptation is a mechanism that lays the groundwork for more advanced techniques like automatic adaptation, e.g., fault tolerance, or fault avoidance. The resurgence of system-level virtualization introduces new avenues to explore dynamic adaptations at the hypervisor level. These mechanisms enable the modification of core capabilities, e.g., scheduler, at runtime without having to tear down all active virtual machines in order to change policies and recompile the hypervisor.

This paper discussed our recent investigations into the Linux implementation of loadable kernel modules. The Linux work, or a subset, may be a potential starting point for adding loadable hypervisor modules to Xen. Additionally, due to its wide usage and recent re-implementation in Linux v2.6, it is a good source for seeing how such mechanism are implemented on a real world system.

We discuss some aspects that would be involved in a similar approach to Linux kernel modules, but for a Xen hypervisor. The key points are access to the hypervisor symbol table, likely from the host OS (domain0), to allow for linking and loading of binary modules. Also mentioned is the relationship between the domain0 and the hypervisor for dynamic allocation and loading of modules at runtime. These dynamic capabilities will require additional state management for tracking the phases of the module.

As highlighted by Teller & Seelam [12] the use of dynamic adaptation is compelling but poses a significant research challenge. As system-level virtualization usage increases the need for runtime customization is likely to become more pronounced. This may be done at the virtual machine level or as discussed here, it could be done at the hypervisor itself.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings*

of the nineteenth ACM symposium on Operating System s Principles (SOSP19), pages 164–177. ACM Press, 2003.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, November 2005.

[3] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44, New York, NY, USA, 2006. ACM Press.

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, third edition, February 2005.

[5] G. Denys, Frank Piessens, and Frank Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, December 2002.

[6] Bryan Henderson. Linux Loadable Kernel Module HOWTO. Linux Documentation Project, August 24, 2006.

[7] John R. Levine. *Linkers and Loader*. The Morgan Kaufmann Series in Software Engineering and Programming. Morgan Kaufmann, first edition, 2000.

[8] Linux 2.6.20 source code, February 2007. http://kernel.org.

[9] Linux kernel website/archive. http://kernel.org.

[10] Rusty Russell. Modules in 2.6: Breaking The Kernel, and What I Learned, 2003. Slides from talk at OSDN Japan - Linux Kernel Conference 2003.

[11] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[12] Patricia J. Teller and Seetharami R. Seelam. Insights into Providing Dynamic Adaptation of Operating System Policies. *SIGOPS Oper. Syst. Rev.*, 40(2):83–89, 2006.

[13] Tool Interface Standards (TIS). *Executable and Linkable Format (ELF): Portable Formats Specification, Version 1.1*.