Facilitating the Development of Soft Devices

Andrew Warfield, Steven Hand, Keir Fraser and Tim Deegan
University of Cambridge Computer Laboratory, J J Thomson Avenue, Cambridge, UK
{firstname.lastname}@cl.cam.ac.uk

1 Introduction

Device-level interfaces in operating systems present a very useful cut-point for researchers to experiment with new ideas. By virtualizing these interfaces, developers can create *soft devices*, which are used in the same way as normal hardware devices, but provide extra functionality in software. Recent years have shown this approach to be of considerable interest: a few examples of block device extension include the addition of intrusion detection systems to disk interfaces [1], the development of "semantically smart" disks [2], and that of time-travel block devices [3]. Other devices, such as network interfaces, have similarly been extended.

Working at the device interface allows an examination of the functional separation between hardware and software: researchers can simulate new features as if they were properties of the device itself. As simple examples, block or network device interfaces might be extended to compress or encrypt data before it is written to disk or transmitted. Alternatively, it may be desirable to prototype entirely new devices in software, bound to existing interfaces, for instance a content-addressable disk.

Unfortunately, researchers face a challenge in extending devices in this manner. Implementors must typically modify an existing operating system to add the new functionality, often by creating OS-specific pseudo-devices. This requirement means learning the OS source and writing scaffolding code to intercept events. Moreover, where new functionality must be developed in-kernel it is difficult to debug and crashes are not contained. Finally, these low-level developments are difficult to share and maintain across systems, as they will be specific to the OS, or even specific version thereof, that it has been developed within.

This paper presents a solution to the problems associated with developing soft devices by extending the existing device interface in Xen [4]. Xen is a virtual machine monitor (VMM) for the IA32 architecture that *paravirtualizes* hardware: Rather than attempting to present a fully virtualized hardware interface to each OS in a Xen environment, guest OSes are modified to use a simple, narrow and idealized view of hardware. Soft devices take advantage of these narrow interface to capture and transform block requests, network packets, and USB messages.

As an initial example of this approach, we have implemented a *block tap*, which is an interface to facilitate the development of soft devices for block device access. The block tap allows soft devices to be constructed as user-space applications in an entirely isolated virtual machine. This strong isolation from the remainder of the system allows a single soft device to work with any OS and hardware available on Xen, and allows developers to work with high-level languages and debuggers. While our approach aims to facilitate development it still provides a high level of performance, sustaining 50MB/s read throughput for disk requests in our experiments.

2 Device Access in Xen

The existing approach to device access in Xen makes use of *split device drivers*, and has been described in detail in [5]. Figure 1 illustrates a split driver: A device driver VM is granted specific access to the physical hardware that it will manage. This VM runs an existing, unmodified (e.g. Linux) device driver to access the device. In addition it runs a *back-end* driver, which provides a simple narrow interface to the device. Operating systems wishing to access the device will use a *front-end* driver, and interact with the back-end over a device channel, which is a shared-memory communications primitive.

This approach aims to improve system stability while still supporting existing device drivers by isolating drivers in a single VM, away from both other OSes and the VMM. Perhaps more importantly though, the split driver interface simplifies device access for operating systems above Xen, as an OS need only implement a single front-end driver to

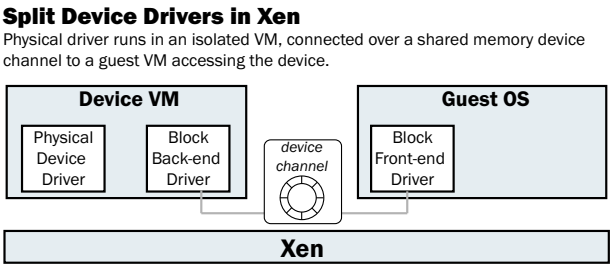

Figure 1: Split device drivers in Xen

support an entire class of devices (e.g. block storage).

Xen's current split block device illustrates exactly how narrow this interface is. The device channel shown in Figure 1 is a single page of memory shared between the two VMs. A bi-directional ring buffer is used to pass messages. Pages of data are attached and mapped separately. The interface has only three commands: `READ`, `WRITE`, and `PROBE`. `READ` and `WRITE` provide block-level access to data, while `PROBE` returns a list of accessible block devices.

3 Implementing a soft device Interface

This section describes how we have extended Xen's existing split device interface to support the development of soft devices. We describe an implementation of a block tap that allows the construction block soft devices. In designing and implementing the block tap, we have attempted to meet three general requirements:

1. **Make device implementation easy.** Our principal goal is to facilitate the development and exploration of new functionality for device interfaces. We have chosen to give developers the option of receiving device requests through a user-level interface in the soft device domain, allowing development in a safe environment with a variety of languages and tools.

2. **Do not modify the existing guest OS or device VM.** While Xen itself achieves performance by modifying the guest operating system, we desire that the soft device interface leave the attached front-end and back-end VMs unmodified. This approach allows the development of soft devices between any hardware and OS combination supported by Xen without necessitating the modification (or even understanding) of that code. Additionally, the soft device interface may be used to trace, debug, or modify an existing split device connection. As such, all soft device implementation exists within an isolated VM using only the shared-memory device channels as an interface.

3. **Maintain satisfactory performance.** We hope to use soft devices in practical situations, and not just as a prototyping tool. By taking advantage of the performance allowed through request batching, we hope to maintain a high level of throughput for soft devices.

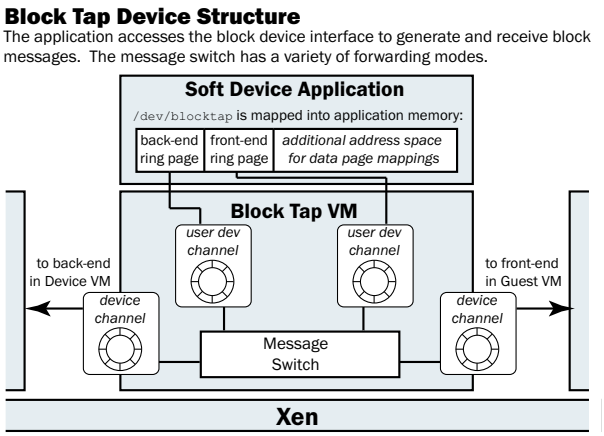

Figure 2: block tap structure.

The remainder of this section describes our block tap implementation addressing these requirements. The block tap is currently about 1300 lines of commented C code, and runs in a Linux-based VM.

3.1 A Switch for Block Requests

The potential ways in which a soft device interface might be used are varied. Developers may desire to simply trace request traffic in order to monitor usage patterns, they may wish to modify in-flight requests, or they may desire to construct a terminating device, which does not forward requests at all.

In order to accommodate these varied modes of operation, we have implemented the soft device interface as a request switch. The driver is plumbed into the device channel between the front-end and back-end domains. In this position, all block requests pass through it.

The new driver acts as a switch, shown in Figure 2, forwarding messages across four rings. Two of these rings are inter-VM shared memory rings as described above. They connect to the front-end and back-end drivers that the soft device interface has been placed between. Two additional rings extend from the block tap up to application space in the same VM. These rings are accessed through a character device that can be mapped by applications.

An `ioctl()` to this character device is used to set the switching mode used for block messages. Three common modes are described here, and shown in Figure 3.

`MODE_PASSTHROUGH` is the lowest-overhead switching configuration. In this mode, messages are passed straight through the driver on to the opposite ring, and completely

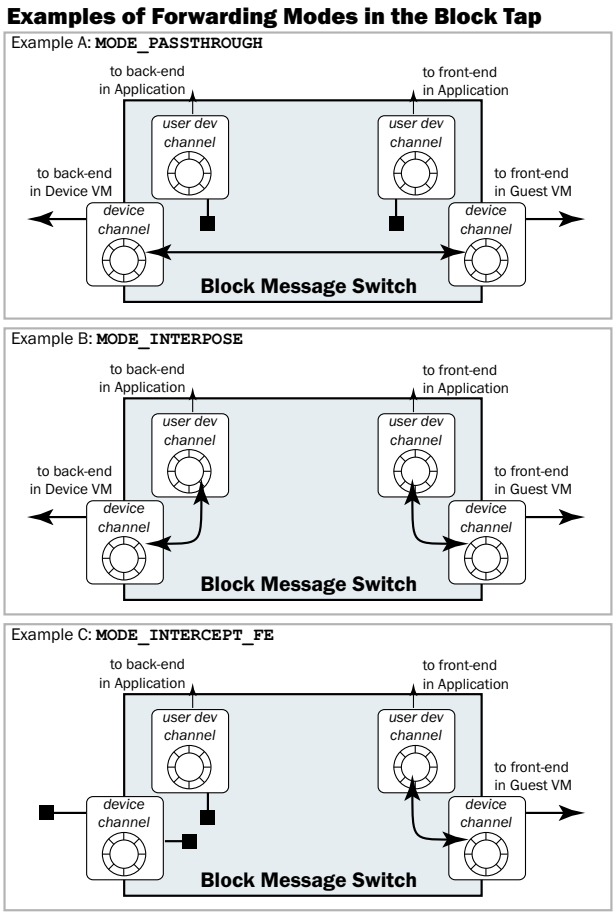

Figure 3: Examples of forwarding modes.

bypass the user rings. Passthrough can be used to implement kernel-level monitoring of block requests, or to implement soft devices in-kernel for improved performance.

`MODE_INTERPOSE` routes all requests and replies across the user rings. An application must attach to the block tap interface and pass messages across the two rings, allowing complete monitoring and modification of the request stream at the application level. This mode can be used to modify in-flight requests, for instance to build a compressed or encrypted block store.

`MODE_INTERCEPT_FE` uses only the front-end rings on the driver, disabling the back-end altogether. This mode allows the construction of full, application-level soft devices, using existing OS interfaces (such as memory, or mounted file systems) as a backing store. This mode can be used to easily prototype new functionality, or to forward block requests to a block device back-end on another physical host (after an OS migration, for instance[1]).

[1]OS migration is feature that we have recently added to Xen, allowing a running OS to move from one physical host to another while executing. One problem which managing migration is that local disks will be left behind.

3.2 The Application Interface

As shown in Figure 2, the user rings are exported to a character device, which is mapped by a library allowing access to the message rings and in-flight requests. Our current implementation allows chains of plugins to be attached to handle block requests. We presently have plugins to provide both copy-on-write and encrypted disks and to allow direct access to image files and remote GNBD disks.

4 Evaluation

Figure 4 shows an analysis of the impact of soft devices on block request performance with respect to both throughput and latency. Tests were performed on a Compaq Proliant DL360, which is a dual Pentium III 733MHz machine with 72.8GB Ultra3 SCSI disks.

Throughput measurements aimed to test the maximum achievable read and write speeds to the local disk. The left graph in Figure 4 shows read and write throughput moving four gigabytes of sequential data to and from disk. The three bars in the graph compare the throughput without using the block tap, using the block tap in `MODE_PASSTHROUGH`, and finally in `MODE_INTERPOSE`. As shown, our soft device interface results in a minimal degradation of throughput. We are capable of achieving 50MB/s read throughput, identical to that achieved by Xen's existing split drivers. On writes, we see about a 15% overhead; we are still investigating the source of this loss of performance.

Latency measures the per-request overhead of synchronous requests to disk. Given that disk requests are heavily batched in general, this is a less meaningful measurement for normal workloads. However, it does represent a worst-case overhead and also gives a clearer illustration of the costs that our implementation imposes. The right graph in Figure 4 shows mean request times across 100,000 4-byte synchronous writes. We see a small overhead in passing requests through the kernel of the virtual device domain in `MODE_PASSTHROUGH`, reflecting the cost of an additional VM context switch and request/response copy[2] in each direction. `MODE_INTERPOSE` is considerably more expensive as it adds two additional context switches and two message copies, in order to pass messages through a user-space application. There are additional costs in mapping attached data pages to user space. However, even this overhead has insignificant impact given the length of average disk seek times. We intend to explore the more demanding performance requirements of network devices in the coming months.

[2]Note that only the request and response structs (respectively 60 and 7 bytes) are copied on the shared memory rings. Pages of data are referenced and mapped separately.

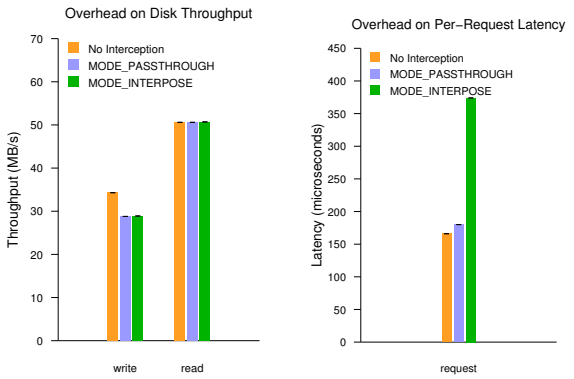

Figure 4: Overhead of virtual block devices.

5 Related and Future Work

The use of virtual machines to provide device extensibility has been explored previously in μDenali [6]. While our intentions of facilitating development of soft devices are identical, our system is very different. μDenali is a VMM hosting BSD VMs, which access devices over micro-kernel-style IPC through Mach's port abstractions. Xen does not use synchronous micro-kernel-style IPC; instead it detaches message transfer (copying/mapping pages between VMs) from notification (virtual interrupts), to achieve high throughput through message batching. We feel that these differences are interesting for two reasons: first, the design presented here represents a considerably different approach to that presented in [6]. Second, as Xen is a publicly available VMM and supports the production use of several popular OSes, we hope that the availability of this work is of general interest to other researchers.

The block tap itself is similar in some ways to the FreeBSD GEOM and Linux EVMS projects, both of which provide a great deal of extensibility the their respective systems' block device interfaces. The block tap aims to provide similar extensibility, but in a virtualized environment, thus catering to a range of operating systems. The use of virtualization also allows strong isolation for soft devices, enhancing stability.

We view the soft device interface as an enabling tool for future work. As mentioned earlier, other projects have explored the use of interposition on device interfaces for such goals as intrusion detection [1], semantically smart disks [2], and fault diagnosis [3]. Two projects that we are particularly interested in exploring over these interfaces in the coming months are described here.

Parallax - a cluster storage system for virtual hosts. Managed virtual machines present different file system requirements than are currently available through distributed or cluster file systems, or network-attached storage. We are currently building a cluster-based storage system to provide high-availability storage for virtual OSes that may migrate between physical hosts, return to arbitrary historical snapshots, and which likely exhibit a large amount of commonality in terms of their system images.

Data-stream watchpoints for pervasive debugging. Interposing on block and network requests will allow existing OS debugger work above Xen to be extended to enable break and watch points on specific content. We intend to allow debugging to be triggered if a specific file is modified, or certain network traffic is sent or received.

6 Conclusion

This paper has briefly presented the implementation of a virtual device interface for OSes on the Xen VMM. Our implementation enables the construction of new functionality to trace, interpose on, or extend existing block device interfaces. We look forward to extending this approach to support network and USB devices, and to building systems above it.

References

[1] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the USENIX Security Symposium.*, August 2003.

[2] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, March 2003.

[3] A. Whitaker, R. S. Cox, and S. D. Gribble. System administration as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, December 2004.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[5] K. Fraser, S. Hand, I. Pratt, A. Warfield, R. Neugebauer, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (OASIS-2004)*, October 2004.

[6] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.