Afterburning and the Accomplishment of Virtualization

University of Karlsruhe, Germany
University of New South Wales and National ICT Australia
http://l4ka.org/projects/virtualization

April 6, 2005

Introduction

The two prominent virtualization technologies, pure virtualization and para-virtualization, provide extremes that may force an undesirable solution. Pure virtualization supports the use of standard legacy operating systems, but with a performance cost, and a substantial engineering cost on some architectures such as x86. Para-virtualization achieves superior performance and scalability, but at the cost of manual modifications to the guest operating systems. The manual modifications of para-virtualization abandon some of pure virtualization's cardinal virtues:

1. Para-virtualization discards the layer of indirection that belongs to pure virtualization, and thus loses the benefit of running on a variety of hypervisors, and restricts the extent of runtime guest mobility.

2. The high-level and manual source code modifications of para-virtualization reduce the trustworthiness of the guest OS, in contrast to the transformations applied by pure virtualization at the instruction set level.

3. Due to the complexity of the modifications to the guest OS, para-virtualization lacks the engineering scalability of virtualization.

Rather than choose between the two extremes, we developed a *pre-virtualization* technique to merge the benefits of pure virtualization and para-virtualization. Pre-virtualization achieves hypervisor diversity and runtime mobility, provides engineering automation, and uses transformations that share the trustworthiness properties of pure virtualization, while retaining the performance characteristics of para-virtualization. Not only is pre-virtualization a substitute for para-virtualization, it enhances pure virtualization environments that recognize a pre-virtualized guest OS.

Pre-Virtualization

Pre-virtualization prepares the OS kernel for execution in a virtualization environment. It locates the *virtualization-sensitive operations* of the OS kernel, and prepares them for replacement with emulation code. The emulation code carries out the *intentions* of the original sensitive operations. Pre-virtualization handles both the sensitive instructions of the processor, and the sensitive memory operations that affect the privileged state of the processor, including page tables and device registers. We base our pre-virtualization solution on an automated, multi-phase process named *afterburning*.

Afterburning

The first afterburning phase applies global, instruction-level transformations to the *sensitive instructions* of the guest OS kernel, at the assembler level where basic-block information is still available. We apply a second phase to detect and transform the *sensitive memory operations* of the guest OS, e.g., those that modify page tables or memory-mapped device registers, and achieve a substantial performance improvement over the first phase alone. The transformations convert the sensitive instructions and memory operations into synchronous function calls to the virtualization layer. The virtualization layer is an *in-place* module, which shares the address space of the guest kernel and exercises loose state consistency, and thus minimizes the number of exits to the virtual machine monitor. The in-place module includes a virtualization kit, offering emulation of the platform devices. Our device emulation is based on transformations to the memory operations that access device registers, avoiding the costs of trapping, permitting us to optimally batch device operations for scalable throughput and latency. The traditional approach to enhancing emulated driver performance, installing custom drivers in the guest OS, is unnecessary in our environment.

Our first two phases achieve compile-time hypervisor diversity; at build time one chooses the appropriate transformations for the target hypervisor. For run-time hypervisor diversity and mobility, we alter the first two phases to generate a single binary compatible with all environments: raw hardware, pure virtual machines, or a subset of para-virtualization environments. Instead of static code transformations, we attach a patch-up table to the binary to support dynamic linking with the hypervisor. The runtime environments apply the third phase, to rewrite the binary, using the annotations contained in the patch-up table. The patch-up table contains sufficient information to locate and transform sensitive operations, and complements both virtualization and para-virtualization environments. For architectures that require extra space for runtime binary rewriting, e.g. IA32, the first two afterburning phases insert innocuous *nop* instructions as place-holders. We further promote runtime diversity and mobility via efficient device emulation; users have less incentive to install hypervisor-specific device drivers in their guest OS's for improving performance. By using standard devices, a guest OS can easily migrate between the diverse virtual machine (VM) environments that publish the same set of standard devices.

Evaluation

We have evaluated our afterburning approach for the Xen hypervisor and the L4 microkernel on IA32, and with an internal hypervisor for Itanium. A single afterburnt Linux 2.6 binary for IA32 can boot on raw hardware (including Intel VT), the Xen hypervisor, the L4 microkernel, and VMware.

We ran the Netperf benchmark, transferring 1 GB of data via Intel gigabit adapters, between a test system running afterburnt binaries and a normal client machine. The test system was a 2.8GHz Pentium 4, and the client machine a 1.4GHz Pentium 4. See Table 1 and Table 2 for the results. CPU utilization was determined with the processor's performance counters (the afterburnt Xen implementation had an incomplete idle loop, preventing measurement of its CPU utilization).

We compared native Linux on raw hardware to the afterburnt Linux on raw hardware, and saw that the additional *no-op* instructions were not a noticeable burden.

We compared a para-virtualized XenoLinux to an afterburnt Linux on Xen, and a para-virtualized L4Linux to an afterburnt Linux on L4. All demonstrated similar performance.

We additionally developed a device emulation layer for the DP83820 network adapter. We configured an afterburnt Linux to use the DP83820 device model, while running as an unprivileged VM on L4, connected to a real e1000 Linux network driver hosted in the privileged VM (see Figure 1). The performance matched that of a customized network driver for virtualizing the connection from the unprivileged VM to the real e1000 Linux driver in the root VM.

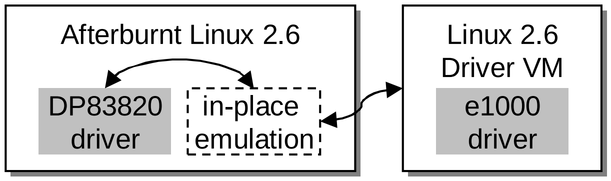

Figure 1: Efficient emulation of the DP83820 network interface, which is routed to the real e1000 device driver.

System	Xput Mb/s	CPU util	cycles per byte
a-L, native	834	28.6%	7.33
v-L, native	827	29.8%	7.69
a-L, Xen	834		
XenoLinux	830	34.3%	8.84
a-L, L4	830	31.3%	8.06
L4L	775	35.0%	9.65
a-L, L4, dev emu	771	49.1%	13.59
L4L, dev emu	772	51.4%	14.21

Table 1: Netperf send performance of various systems. Legend: "v-L": native Linux, "a-L": afterburnt Linux, "L4L": para-virtualized Linux on L4, "XenoLinux": para-virtualized Linux on Xen, "dev emu": device emulation in-place, interfacing to device driver in separate VM. The column "cycles per byte" represents the number of non-idle cycles necessary to transfer a byte of data, and is a single figure of merit to help compare between cases of different throughput.

System	Xput Mb/s	CPU util	cycles per byte
a-L, native	712	31.5%	9.46
v-L, native	713	33.0%	9.90
a-L, Xen	711		
XenoLinux	711	39.0%	11.72
a-L, L4	709	37.1%	11.17
L4L	712	35.7%	10.71
a-L, L4, dev emu	707	59.8%	18.06
L4L, dev emu	708	60.3%	18.21

Table 2: Netperf receive performance of various systems. See Table 1 for the legend.

Conclusion

Pre-virtualization is a significant improvement over standard para-virtualization: it offers competitive performance, while adding engineering scalability, trustworthiness in the integrity of the guest OS, and hypervisor diversity. The adoption of pre-virtualization would help to quickly achieve virtualization for more processor architectures, and to support everyone's favorite hypervisors.