Gelato@UNSW

Performance and Scalability on Itanium

www.gelato.unsw.edu.au

[Para]Virtualisation Without Pain

Peter Chubb

Reporting on work by

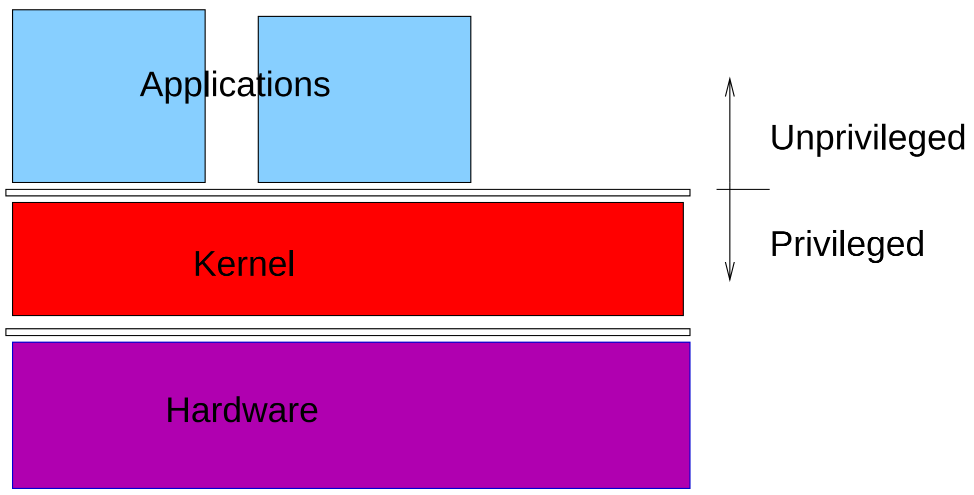
Matthew Chapman Myrto Zehnder myself

Gelato@UNSW

National ICT Australia
and
The University of New South Wales

January 2007

Normal Machine Operation



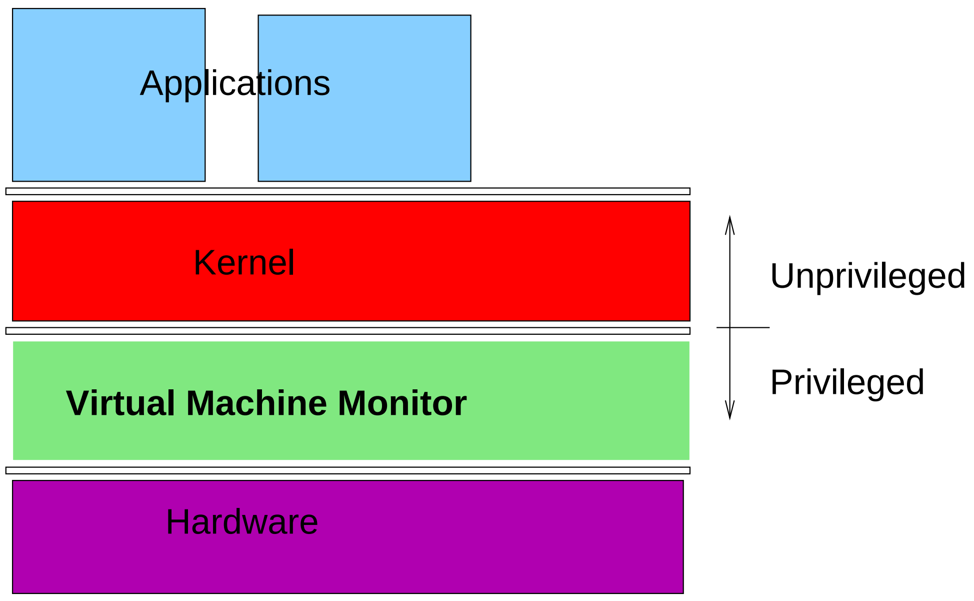
Normally, user space programs run along happily without caring much about the state of the hardware supporting their operation. If they want the hardware to do something (like give them some more memory, or some data from a file) they invoke the kernel by means of a *system call*. The kernel runs with a higher privilege level, and controls access to the hardware — it also contains all the code that understands how to drive devices.

Sometimes, however, you want to run more than one instance of an entire operating system. Typical reasons are:

- 'Virtual Hosting', where multiple complete images run simultaneously on the same hardware. Providing the isolation between the virtual machines is sufficiently complete, sharing resources like this can be more cost effective (in sysadmin time, airconditioning, etc) than providing a separate machine for each use.

- For security. For example on an embedded system such as a mobile phone, one might wish to run the code that controls the radio transmitter (and is highly regulated by government) in a completely isolated virtual machine.
- For experimentation/development of new kernel features — it's easier to do this when not on the bare metal (reboot, for example is a *lot* faster when the BIOS doesn't have to check everything under the sun).
- To allow more predictable real time performance (e.g., the ADEOS approach)
- etc., etc.

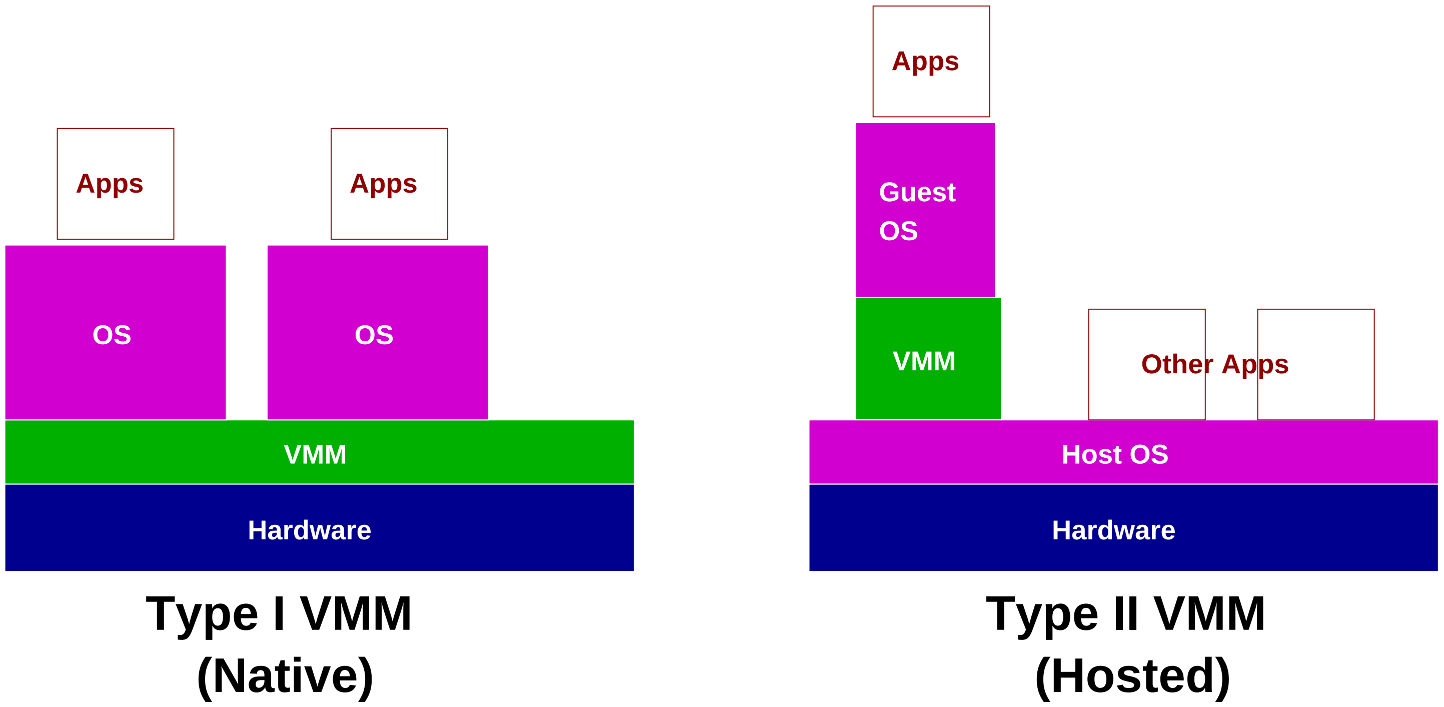
Virtualised Machine Operation



One approach used is to deprivilege the operating system kernel. Then when the kernel tries to do something special (like talk to the hardware), its attempt is trapped into a *Virtual Machine Monitor* (VMM), *supervisor* or *Hypervisor*. The VMM pretends to be a machine that looks more-or-less like the real one, while controlling page table, direct device and privileged system control register accesses.

A bit of terminology: the operating system running on the virtual machine is often termed a *guest*; the VMM is sometimes termed a *host*.

Virtual Machine Monitors



There are two kinds of virtual machine monitors (VMM). A *Type I* VMM controls the hardware directly (The nomenclature was originally introduced by [Goldberg, 1973]). Guest operating systems see a virtualised version of the hardware. Xen [Barham et al., 2003] and vNUMA [Chapman and Heiser, 2005] are examples of Type I VMMs.

A *Type II*, or *hosted* VMM runs as an application on a normal operating system. Examples are UML [Dike, 2000], VMware Workstation [Sugerman et al., 2001] and Gelato@UNSW's own LinuxOnLinux [Chubb, 2

Both a type I and a type II VMM have to virtualise not only the processor(s), but also some kind of I/O system. Most commonly, devices are emulated at 'standard' I/O addresses — for example, the free VMware workstation emulates a standard IDE controller based on the Intel PIIX4 chipset, a VGA controller, and a LANCE ethernet. The alternative is to paravirtualise disk and console acess, as vNUMA does using the SKI

simulator Supervisor System Calls to emulate a SCSI disk and ethernet controller.

The Core Virtualisation Problem

- ✔ Performing a privileged operation traps
- ✘ Reading system state doesn't always trap
- ✘ System and User state not cleanly separated
- ✘ Trapping is s l o w.

If one runs an operating system kernel (e.g., Linux) in deprivileged mode, then attempts to access device registers, or to change interrupt collection state, etc., (all privileged operations) will trap to the VMM.

The VMM then emulates the operation in the virtual processor, and resumes the kernel.

Unfortunately, not all operations that one would like to trap do so; and some instructions don't trap, but behave differently in different privilege levels.

Moreover, trapping to the VMM can be very expensive — certainly much more expensive than the operation performed. In a type-II VMM, trapping usually involves a signal and a context switch; with Linux as host, which does not (yet) have the ability to perform a directed context switch, if the machine is at all busy context switches can add extra latency (a trap merely makes the VMM runnable, which puts it onto the run queue to contend with all other processes for processor time).

Architecture Extensions for Virtualisation

- *Vanderpool* extensions:
  - add *extra privilege level*
  - *All* changes to privileged state trap

Intel in recent processors introduced specific support for virtualisation, called variously the *silvervale* or *vanderpool* extensions. These extensions add an additional privilege level; if the extension is enabled, operations that need to trap for effective virtualisation *do* trap — to code running at the most privileged level.

This solves the system state mixed with non-system state problem, at least for type-I hypervisors (and with some kernel support, for type-II VMMs as well; this support has very recently come into Linux with the `CONFIG_KVM` option.)

Virtualising Itanium

- Easier than IA32
- but non-trivial
- Non-virtualisable elements include:
  - `cover` modifies IFS register when IC off.
  - `thash` and `ttag` reveal real, not virtualised pagetable details

To virtualise an architecture requires

1. Clean separation between user and system state
2. All instructions that modify system state need to be privileged
3. All system state has to be visible

Itanium is not fully virtualisable, although maybe the Silvervale extensions available in Montecito will fix this.

At present, the `cover` instruction (which creates a new empty stack frame) is not privileged; nor does it need to be. However, if it is executed with interrupt collection off, it as a side effect saves information into the *interruption function state* (`IFS`) register. The side effect has to be

emulated by the virtual machine, but as the instruction does not trap the VMM doesn't know it needs to.

Likewise, the instructions for calculating the hash and tag of a virtual address in the VHPT are not privileged. But they return not the virtualised address and tag, but the one of the underlying real machine.

Virtualisation Techniques

- Full virtualisation
- Paravirtualisation
- hybrid techniques

Because of the problem of non-trapping instructions, almost every VMM modifies its guest, replacing sensitive non-trapping instructions with sequences that trap.

One easy technique is to replace all non-trapping instructions with trapping instructions, often illegal operations. This solves the non-clean separation problem, but not the performance.

Another technique is to replace all privileged operations in the guest operating system with instructions that alter the state in a memory region shared between VMM and guest — the virtual CPU state. Privileged operations and attempts to access non-accesible memory still trap, but simple operations (like changing an interrupt mask) are almost free.

In addition, such a *paravirtualised* operating system can have special *VMM system calls* added to it, to allow the hypervisor to perform complex actions in one chunk rather than having to infer a complex action from the sequence of privileged operations performed. For example, context switching in the guest operating system on Itanium sets four region registers (RRs), each of which could cause a trap and return; it'd be much simpler to tell the VMM that the context is about to change and to provide all the RRs at once.

Hybrid Virtualisation Techniques

- *optimised* paravirtualisation
  - Fully virtualise but...
  - paravirtualise non-trapping instructions
  - then paravirtualise performance hotspots
- Binary rewriting e.g., VMware
  - replace non-trapping instructions with trapping ones
  - Maybe paravirtualise for performance as well
- Our approach ...

Xen on Itanium currently uses a technique called *optimised paravirtualisation*. Most of the kernel is unchanged, except for the bits necessary for correctness. In addition, after profiling and performance measurement, other code paths are paravirtualised to remove bottlenecks.

Paravirtualising Itanium

- Provide (overridable) *access functions* for privileged operations.
  - ✔ (already done to support *icc*)
  - ✘ Assembly language files not done
- Replace key routines (e.g., to set up page tables) with calls to hypervisor.

To paravirtualise the system, one starts by finding all the privileged and should-be-privileged instructions, and replacing them with calls to the hypervisor. IA64 Linux already wraps these instructions in macros so that they can be used from C code. Finding all the instances in the assembly level code is more interesting. This all starts to feel like a lot of work, especially if the changes have to be pushed upstream to a community that, until recently, didn't much care about virtualisation.

For full paravirtualisation, one would make more extensive changes — adding what are essentially hypervisor calls to tell the VMM about state changes that it is (or ought to be) interested in — for example, changing PTEs.

Why paravirtualise by hand?

The assembler already has to be able to find all the instructions we're interested in. Why not just it to find and fix the instructions? And then add performance optimisations later. This idea came from the University of Karlsruhe, see [LeVasseur et al., 2005], who did the work for IA-32; we at UNSW worked on IA-64 with slightly different focus.

Previrtualisation

- Replace `as` wth a `perl` script
- Script rewrites instructions, then invokes real `/usr/bin/as`
- Script saves addresses in special ELF section

Rather than changing the assembler itself (and thus having to track binutils development) we chose to write a little perl script that runs instead of the assembler, and then invokes the assembler on a modified version of the input file.

The modifications include rewriting instructions, and creating a special ELF notes section containing a table of addresses of rewritten instructions.

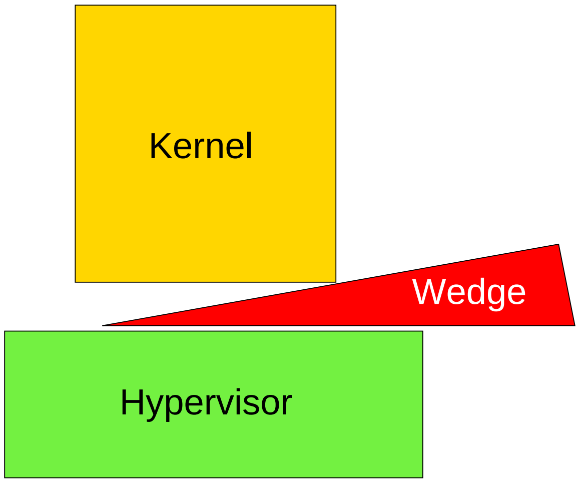
Instruction Rewriting

Original	**Rewritten**	**Loaded**
`rsm.i psr.ic`	`rsm.i psr.ic`	`mov  r5=__afterburn_cpu+`
	`nop`	`mov.l r6=[r5]`
	`nop`	`dep r6=0,r6,13,1`
	`nop`	`mov [r5]=r6`

The three columns in the slide show the original, the version as rewritten by the assembler, and the version as rewritten by the loader. More complex instructions require more work,of course, and may be implemented by a hypervisor system call.

The nice thing is that the rewritten code will still run on the bare hardware.

The *wedge*



Having rewritten the code, there's still need to interface with a variety of hypervisors. This is done by means of a *wedge*, a piece of code that can be called by the rewritten code to interface into the hypervisor.

A different wedge is needed for each combination of operating system and hypervisor. Wedges are not particularly large — the x86 XEN wedge is around 5000 LOC (including comments and whitespace).

In addition, it's possible to increase performance by hooking particular operations, in a way similar to manual paravirtualisation. For example, telling the hypervisor about pte changes directly, instead of allowing it to infer them.

The result is pretty good.

Using the XEN hypervisor on Linux IA64, the automatically paravirtualised code is very close to the manually paravirtualised code, with a fraction of the engineering effort, and almost no changes to the source tree. This work was carried out by Matthew Chapman.

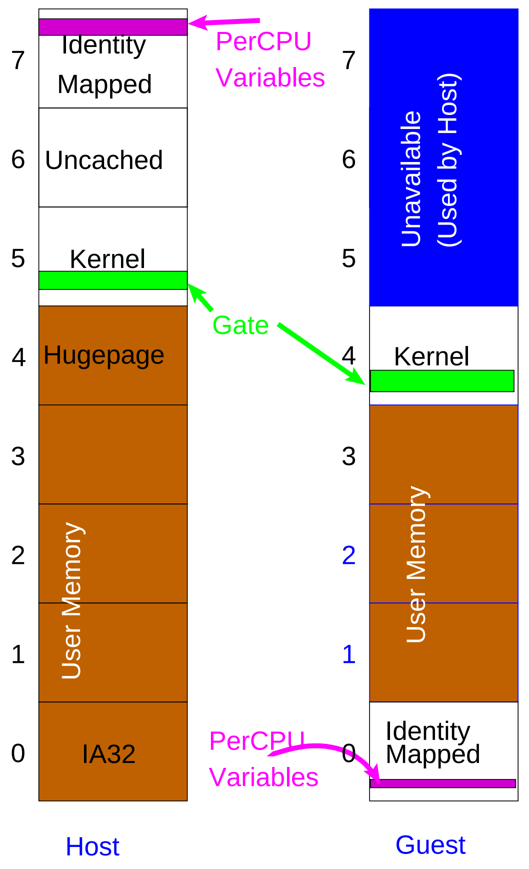
	2p/0k ctxsw (usec)	Pipe (usec)	AF UNIX (usec)	TCP (usec)	File reread (MB/s)	Mmap reread (MB/s)
Native	1.270	4.197	7.89	13.7	2273.5	889.1
Manual	2.720	6.68	11.6	18.8	2200	879
Auto	2.530	6.84	11.9	17.3	2200	879

In this slide, ‘Native’ is the standard Linux kernel; ‘Manual’ is the manually optimally-paravirtualised Xen guest, and Auto is the guest paravirtualised by afterburning.

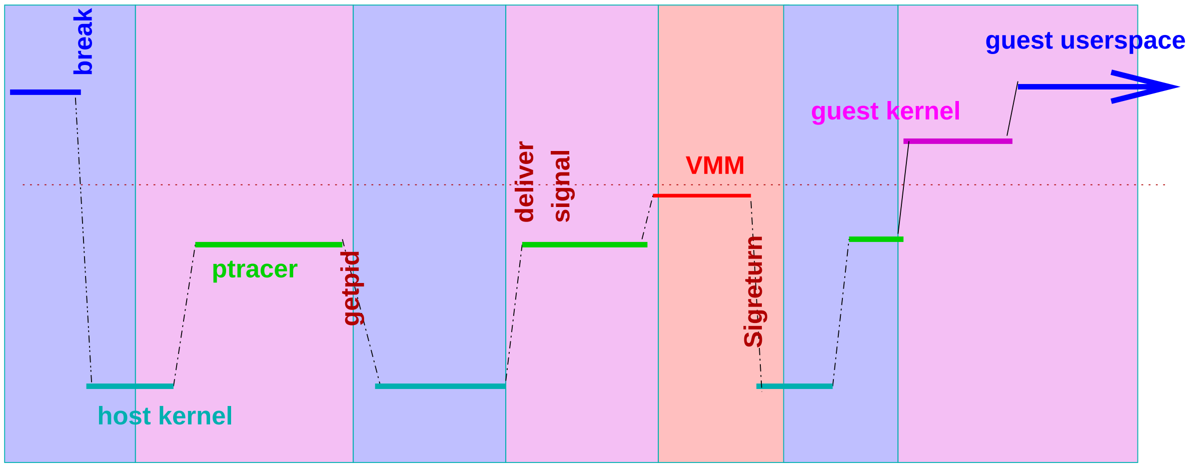
Type II VMMs on Linux

- UML on IA32 — Heavily paravirtualised
- On Itanium:
  - Rearrange memory map to avoid host
  - per-CPU data area moves to region 0
  - Reuse device etc., infrastructure for SKI Simulator
- System calls via `break` a problem

You can use Linux itself, together with a bit of extra software, as a virtual machine monitor. On Itanium, this requires significant changes to the base kernel, as the host kernel steals regions 5, 6 or 7, making them unavailable to any guests. In addition, hugeTLBFS cannot be configured in either host or guest, as that takes away region 4, and Linux needs at least two regions for the OS and the hypervisor, leaving only two for the guest's user-level programs.



In the guest, the per-cpu region, the kernel's identity-mapped region, the VMM text and data, and a file representing 'physical' memory are all mapped into region zero. The kernel's virtual addresses are in region four, because there are several places in Linux where it is assumed that all kernel addresses are above all user-space addresses. While this reduces the space available for user processes, most processes (other than those running in IA-32 emulation mode or that use hugeTLBfs) only use regions one through three anyway.



The main performance issue with using Linux as the VMM involves system calls. In Itanium Linux, there are two ways to invoke a system call. The old way is to use a `break` instruction, which traps to the kernel; the new way is to branch to an entry point in a shared gate page.

If a program running under a guest operating system does a system call via `break`, then the host kernel intercepts it and tries to implement it as a system call. But what we want is for the guest OS to run it.

The current solution is for the VMM to set up a separate process, that uses *ptrace* on the VMM plus guest OS process, and intercepts all system calls, redirecting them to the guest OS if and only if they were not executed from the VMM itself. This leads to a major performance loss, as every address space switch causes many VMM system calls, and every signal delivery not only stops the machine and transfers control to the ptracer, but causes two more stops in `sys_rt_sigreturn()`.

Hack the host

- Restrict ptrace to ranges of addresses
- Add **`PT_ONESHOT`** flag
- Add **`PT_NOSIGSTOP`** flag
- Add way to set `psr.dfh` (necessary for correctness)

To avoid these superfluous context switches, I added a heap of hacks to the host operating system. The first one I tried was to restrict ptrace's stops to only the addresses we're interested in. Unfortunately when a process returns from a signal handler it does so via a system call, **`sys_sig_rt_return()`** which essentially does a **`setcontext()`** — and ptrace will stop the traced process for each signal return. So I added **`PT_ONESHOT`**, which disables a single ptrace, and arranged for it to be set in the **`sys_rt_sigreturn`** path. The large number of signals was still slowing things down, so the next step was to turn off ptrace stopping with signals.

The remaining hack was to allow the virtual machine monitor to turn of the DFH bit in the PSR if it was the current owner of the FPU, so that the floating point state could be saved/restored appropriately.

The hacks reduce the overhead in the trace thread from around 50%

to around 5%. But the resulting virtual machine still feels slow and sluggish.

To deliver a **`SIGILL`** still takes three context switches. And there are a lot of them. Paravirtualisation by afterburning seems the ideal solution to remove the extra context switch overhead. And maybe we can remove some other overheads at the same time, by inlining common operations.

Afterburning

- For now, link wedge with OS
- Hacks to guest prevent use on bare metal or simulator
- Development harder than it might be
  - At least until the Afterburner and Wedge are bug-free
- 5 days part time (4 h/day) work!!!!

Some Linux-on-Linux results

	2p/0k ctxsw (usec)	Pipe (usec)	AF UNIX (usec)	TCP (usec)	File reread (MB/s)	Mmap reread (MB/s)
L-o-L	252	490	859	540	1684	886
virt	1550	3071	5100	4060	185	887
Native	1.270	4.197	7.89	13.7	2273.5	889.1
Manual/Xen	2.720	6.68	11.6	18.8	2200	879
Auto/Xen	2.530	6.84	11.9	17.3	2200	879

As can be seen, changing to afterburning cuts costs dramatically compared with full virtualisation; but a type-I hypervisor (or the bare metal) is still much faster. ‘virt’ here is a fully virtualised system that merely replaces non-trapping sensitive instructions with trapping instructions; ‘L-o-L’ is the afterburnt system with the ptrace hacks in place.

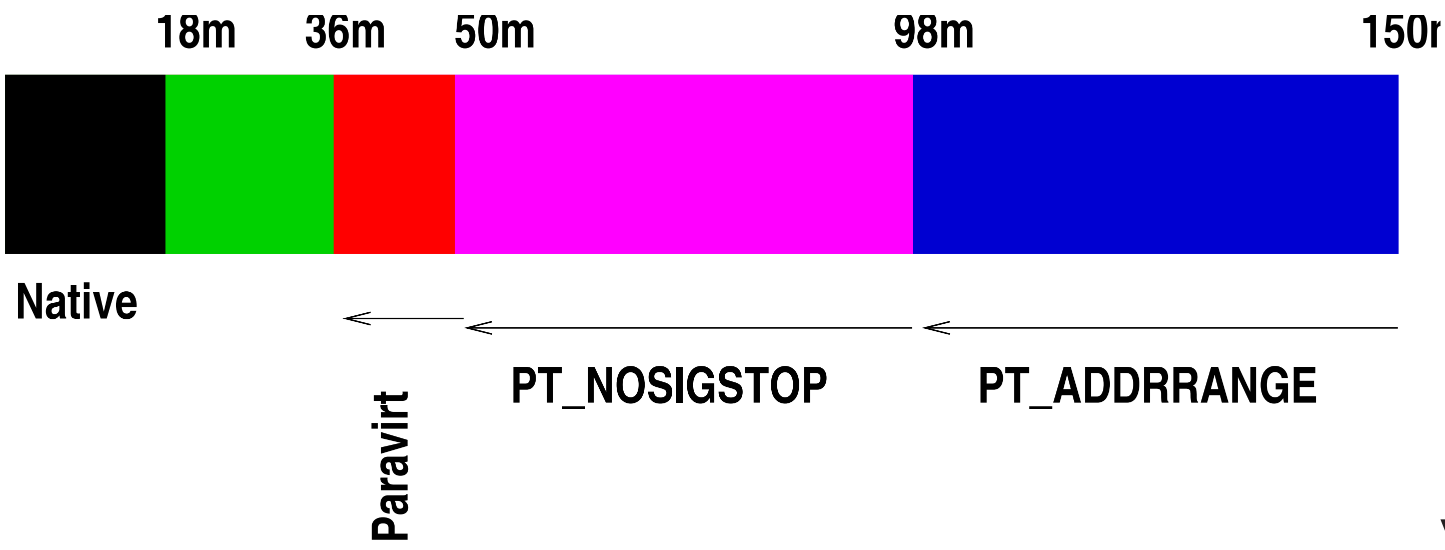
- Better than before —But still too slow!
- Main overheads are system call, I/O and context switch time.

There’s still room for improvement.

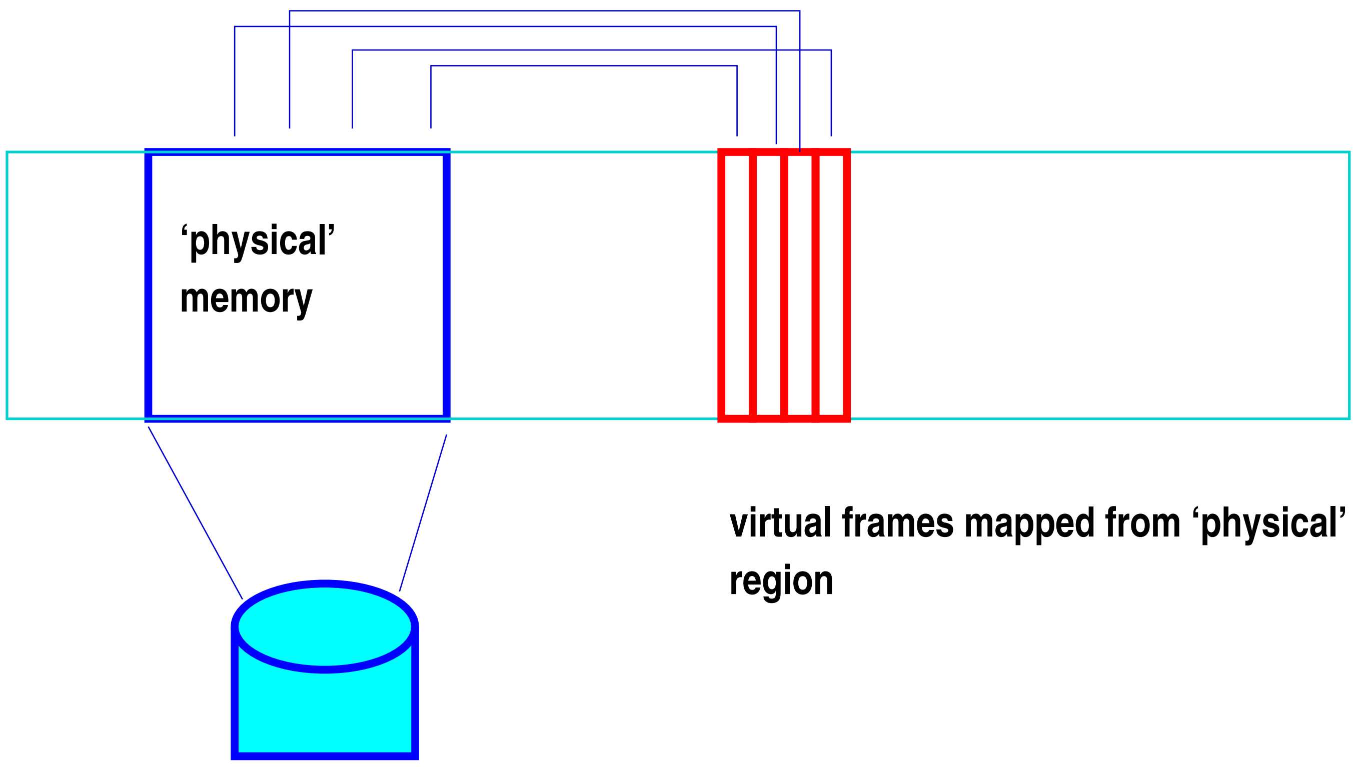
More improvements

- Fast system calls
  - 4 fewer context switches per syscall — ptracer mostly unused
  - Except for `clone`, `fork` etc
  - And legacy statically linked programmes
- VMM improvements
  - Eager mapping of address spaces cuts 30% off context switch time
- Direct access to hardware (using user driver framework)

The obvious steps are first to get fast system calls working. A fast system call is entered by branching to the gate page, and never involves the host kernel at all. So the ptracer overhead (or more importantly the context switches involved in switching to and from the ptracer) is not incurred.



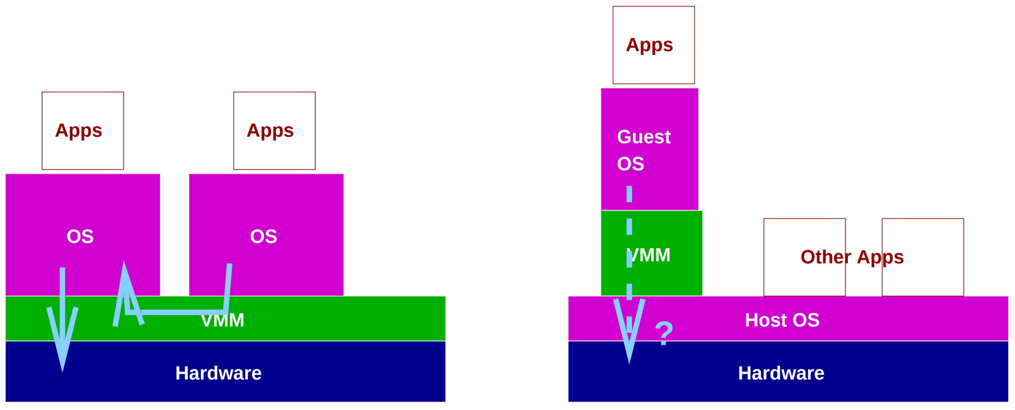
With faster system calls, context switch overhead becomes the main problem.



The VMM has an area of memory shared between virtual CPUs to act as its physical memory. Pageframes from the ‘physical’ memory are mapped into the virtual address space of the guest under control of the guest OS.

Every time the address space changes, the VMM has to unmap all the mappings for the current address space. Originally it was lazy, and waited for the guest to fault in new mappings; changing to keep a few contexts around and eagerly map them shaved 30% off the context switch time.

Virtual I/O



Type I VMM
(Native)

Type II VMM
(Hosted)

Xen allows device drivers in dom0 to access real hardware, and perform I/O on behalf of other guests on the system. Can we do the same for a Type-II virtual machine monitor?

- Use User-Level Driver framework developed at UNSW
- Device Discovery, IO-space access, Interrupts, DMA

We have been working on deprivileged device drivers (see [Leslie et al., 200 and [Chubb, 2004]) for some time. The framework developed here allows suitably privileged processes to set up and tear down mappings for DMA, and to receive interrupts.

It should be possible to integrate this into the VMM, and respond to requests by the guest kernel to do, say, **`pci_dma_map_sg()`** with an appropriate call into the user level driver framework.

Myrto Zehnder did this in early ’06, leading to a heavily hacked system where the user-level guest kernel could access an IDE PCI controller as if it were on the bare metal. Unfortunately the work she did was fairly restricted to that one device.

Remaining to do are implementing a virtual PCI bus, providing enough ACPI tables to discover it, and generalising the infrastructure to allow access to any device that is not already claimed by the host.

Future Work

- Gate page for VMM calls
  - Reduce cost of many ops currently using **`break`**
- Finish device driver work

The remaining pieces of work to improve usability and performance, are to remove the use of break for hypervisor calls, and use a gate page instead. I do not expect that top lead to a massive performance enhancement, as system time in the guest is already fairly low.

Also, I'm intending to work on removing the need for a ptracer process, which will allow gdb to be used directly on the guest (as well as improving performance); and on the ability to multiplex address spaces for a single process in the host, thus speeding context switch time immensely.

References

[Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003).

Xen and the art of virtualization. pages 164–177, Bolton Landing, NY, USA.

[Chapman and Heiser, 2005] Chapman, M. and Heiser, G. (2005). Implementing transparent shared memory on clusters using virtual machines. pages 383–386, Anaheim, CA, USA.

[Chubb, 2004] Chubb, P. (2004). Get more devices drivers out of the kernel! In *Ottawa Linux Symposium*, Ottawa, Canada.

[Chubb, 2005] Chubb, P. (2005). [Para]virtualisation without pain. Gelato ICE conference, Brazil.

[Dike, 2000] Dike, J. (2000). A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, USW.

[Goldberg, 1973] Goldberg, R. P. (1973). Architecture of virtual machines. In *AFIPS*, pages 74–112, New York.

[Leslie et al., 2005] Leslie, B., Chubb, P., Fitzroy-Dale, N., Götz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y. R., Elphinstone, K., and Heiser, G. (2005). User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664.

[LeVasseur et al., 2005] LeVasseur, J., Uhlig, V., Chapman, M., Chubb, P., Leslie, B., and Heiser, G. (2005). Pre-virtualization: Slashing the cost of virtualization. Technical Report PA005520.

[Sugerman et al., 2001] Sugerman, J., Venkitachalam, G., and Lim, B.-H. (2001). Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. Boston, MA, USA.