[Para]virtualisation without pain

Peter Chubb

Gelato Project

National ICT Australia

and

The University of New South Wales

October 2005
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 better real time performance (e.g., the ADEOS approach)

• etc., etc.
One approach used is to deprivilege the operating system kernel. Then when the kernel tries to do something (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 and direct device 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.
If one runs an operating system kernel (e.g., Linux) in deprivileg ed 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.

Trapping to the VMM can be very expensive — certainly much more expensive than the operation performed.

An alternative is to replace all privileged operations in the guest op- erating 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-accessible 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 com-
plex 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.
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 (and I believe that this is the case even with the Silverdale extensions available in Montecito).

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.
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, by and large, doesn’t 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 use it to find and fix the instructions?
Previrtualisation

- Replace `as` with 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.
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.
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.
<table>
<thead>
<tr>
<th></th>
<th>2p/0k (usec)</th>
<th>Pipe (usec)</th>
<th>AF UNIX (usec)</th>
<th>TCP (usec)</th>
<th>File reread (MB/s)</th>
<th>Mmap reread (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1.270</td>
<td>4.197</td>
<td>7.89</td>
<td>13.7</td>
<td>2273.5</td>
<td>889.1</td>
</tr>
<tr>
<td>Manual</td>
<td>2.720</td>
<td>6.68</td>
<td>11.6</td>
<td>18.8</td>
<td>2200</td>
<td>879</td>
</tr>
<tr>
<td>Auto</td>
<td>2.530</td>
<td>6.84</td>
<td>11.9</td>
<td>17.3</td>
<td>2200</td>
<td>879</td>
</tr>
</tbody>
</table>

**Linux as a VMM**

- **UML on IA32** — Heavily paravirtualised
- **On Itanium:**
  - Regions 5–7 reserved for host OS
  - per-CPU data area moves to region 0
  - Region 0 identity mapped (old region 7)
  - Region 4 becomes Gate and virtual Kernel (old region 5)
  - 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.

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, 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 cause two more stops in the sys_rt_sigreturn() path.
Hack the host

• Restrict ptrace to ranges of addresses
• Add PT_ONESHOT flag
• Add PT_NOSIGSTOP flag
• Add way to set psr.dfh

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 off 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.
Results

- Still too slow:
- Signal delivery and return is slow

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!!!!

<table>
<thead>
<tr>
<th></th>
<th>2p/0k</th>
<th>Pipe</th>
<th>AF</th>
<th>TCP</th>
<th>File</th>
<th>Mmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctxsw (usec)</td>
<td>252</td>
<td>490</td>
<td>859</td>
<td>540</td>
<td>1684</td>
<td>886</td>
</tr>
<tr>
<td>virt</td>
<td>1550</td>
<td>3071</td>
<td>5100</td>
<td>4060</td>
<td>185</td>
<td>887</td>
</tr>
<tr>
<td>Native</td>
<td>1.270</td>
<td>4.197</td>
<td>7.89</td>
<td>13.7</td>
<td>2273.5</td>
<td>889.1</td>
</tr>
<tr>
<td>Manual/Xen</td>
<td>2.720</td>
<td>6.68</td>
<td>11.6</td>
<td>18.8</td>
<td>2200</td>
<td>879</td>
</tr>
<tr>
<td>Auto/Xen</td>
<td>2.530</td>
<td>6.84</td>
<td>11.9</td>
<td>17.3</td>
<td>2200</td>
<td>879</td>
</tr>
</tbody>
</table>
As can be seen, changing to afterburning cuts costs dramatically compared with full virtualisation; but a different hypervisor (or the bare metal) is still much faster.

**Future Work**

- Get fast system calls working
- Fix $r^4\ldots r^7$ save/restore
- Gate page for VMM calls
- Reuse UML features in VMM
- User-level drivers in guest