



Work in Progress at UNSW
Memory Management and Device Drivers

Peter Chubb

(and a cast of thousands)

Gelato Project

National ICT Australia

The University of New South Wales

25 May 2004

The Device Driver Problem

- Lots of code.
 - 1.9 Mloc (drivers)/4 Mloc (total) (2.6.6)
- Tend to be buggier than mainline code
 - Estimated 85% of bugs in drivers

The drivers subdirectory contains around 1928595 SLOC of code; the files in all of Linux contain around 4,026,139 lines of code, as measured in 2.6.6 using David A Wheeler's SLOCCount utility. These figures are indicative only: some things in the `drivers` directory aren't drivers (e.g., the block layer infrastructure), and some things outside the `drivers` tree are drivers (e.g., in the `sound` tree).

Chou (2001) says that more than 85% of bugs are in driver code; other studies show similar results.

The people who want to write drivers are typically not kernel programming experts. They're people who've manufactured or bought a device, and want to have it working in Linux. As such, the code quality of the average device driver is poor.

Churn

- Drivers use kernel internals.
- Kernel internals change *fast*
- Not everyone cares about every device...
- So drivers aren't updated.
 - 82 drivers marked 'BROKEN' or 'BROKEN_ON_SMP';
 - more cannot be unloaded...
 - or simply don't work.

Traditional drivers are exposed to changes in the rest of the kernel. Kernel developers like changing kernel code, 'to make it better'.

Keeping driver code up-to-date is a major pain. Most people can't test all the drivers — they just don't have the hardware. So fixes are made to 'get it compiled'; whether or not it's the *right* fix has to wait until some poor person tries to use it.

(un)safety

- Drivers can access all memory
 - Faults in drivers can crash/corrupt system

On the other hand, the kernel is exposed to a driver. I've already asserted that drivers tend to be buggier than core kernel code (they're less exercised, and often written by less experienced people), so confining faults to the faulting driver is an important issue.

In-kernel programming is *Hard*

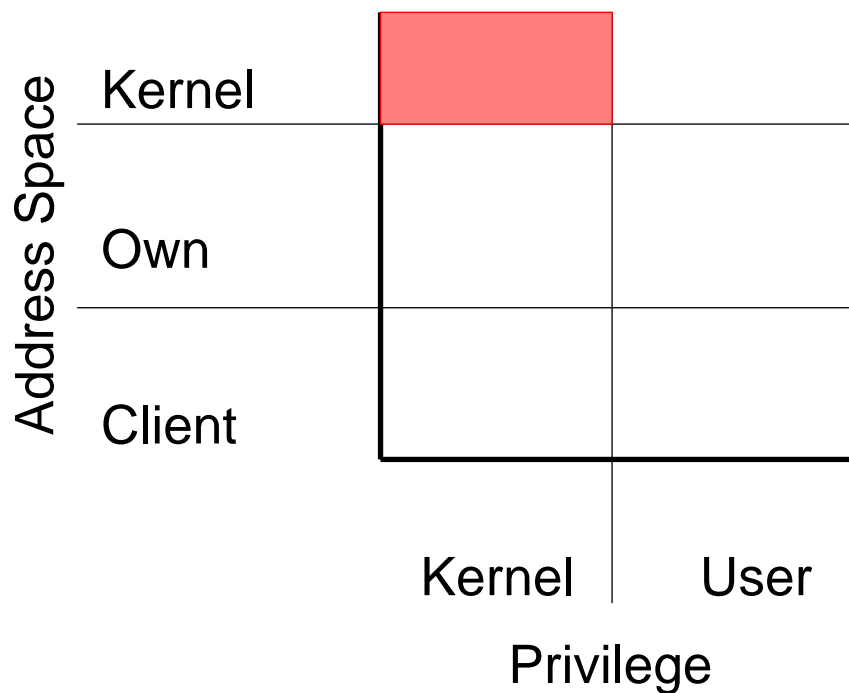
- Kernel programming:
 - Must be C
 - Locking, Preemption, interfaces, idioms
- Reboot cycle slow
- Normal debuggers not available
 - kdb and kgdb now available for some kernels on some platforms (Not 2.6 IA64)
 - *really* need serial console

Programming in the kernel is harder than programming in user space. Most standard library functions aren't available, or have slightly different semantics; you have to worry about (and preferably test on) multiple platforms with different configuration options; and in-kernel you have to worry about interrupts, locking and preemption.

Moreover the problem of feature-churn is always present. What works in a 2.4.20 kernel won't work in a 2.5.9 kernel; what worked in 2.5.9 may not work in 2.6.6.

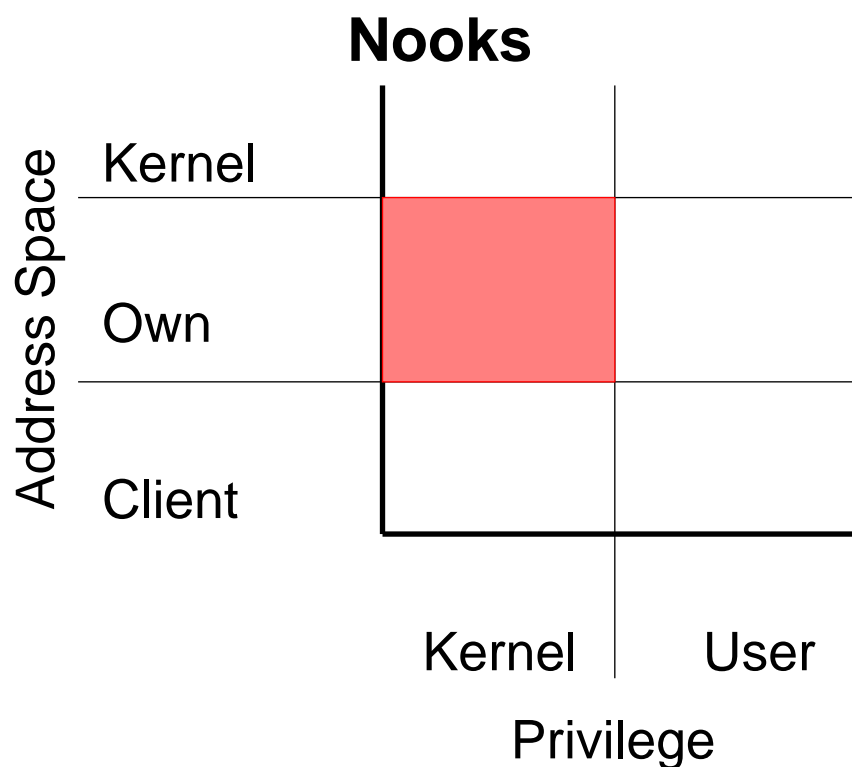
Alternatives

Traditional Driver



Traditionally, device drivers are in the kernel's address space, and run with kernel privilege. As such they have full access to all the kernel features. Put another way, the kernel is fully exposed to driver bugs, and the driver is fully exposed to kernel churn.

Alternatives

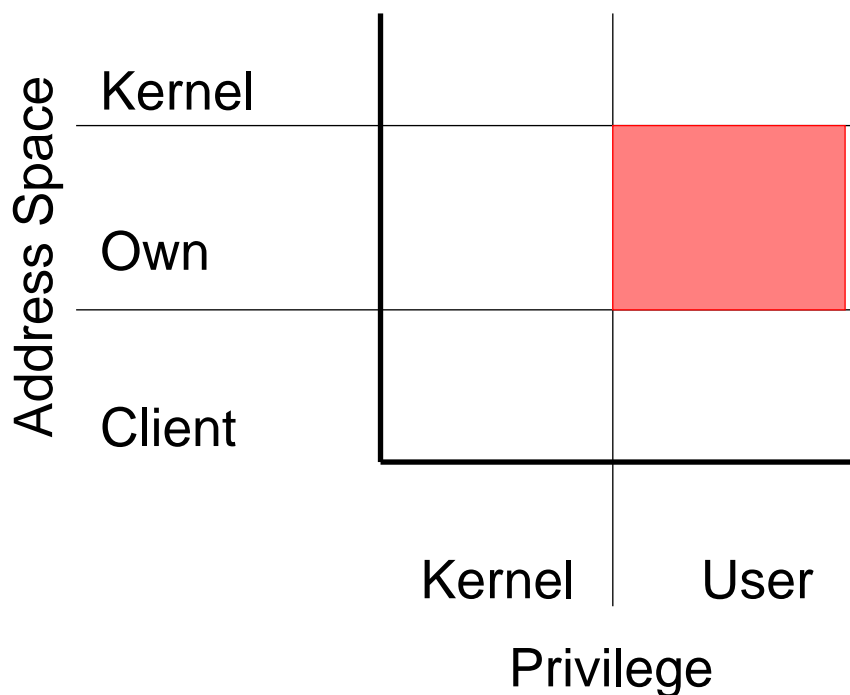


The Nooks (Swift, 2002) work builds a light-weight protection domain inside the kernel for each driver. This has the advantage that almost-unmodified drivers can be used (thus leveraging the enormous amount of work that's already gone into the in-kernel drivers) while isolating the drivers in their own address spaces. The work as yet has been done only for uni-processor IA32; it's unclear to me how much work is involved in ports to other platforms and to SMP.

This work involved creating stubs to each kernel interface function used; unfortunately, most drivers use a lot of the kernel's internal functionality.

Alternatives

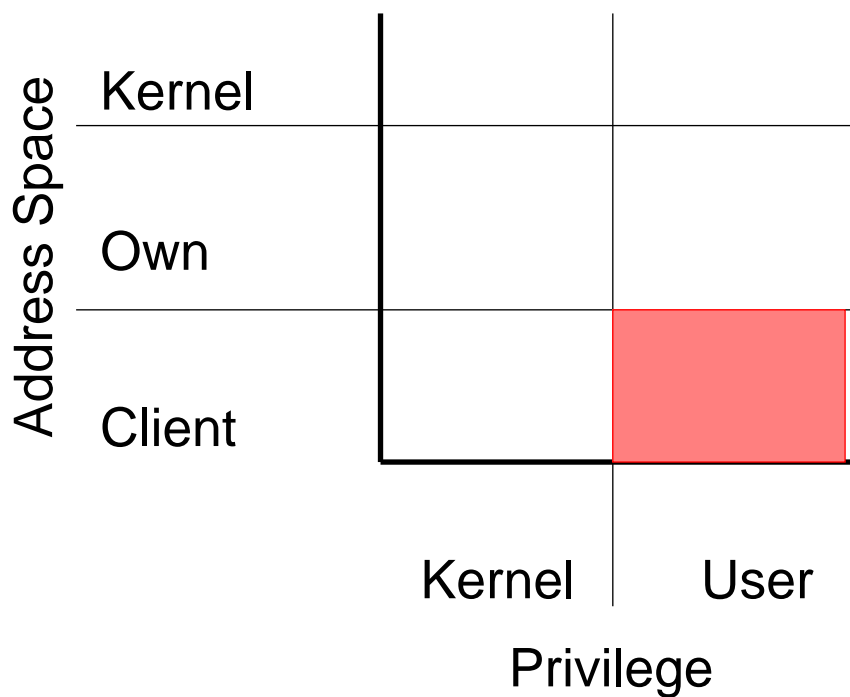
Separate User Process: e.g., Xfree86



Drivers that don't need DMA or interrupts, can be put into user space, in their own address space. Clients can then talk to the driver using a client/server model. Xfree86 is a prime example of this approach; it however does use kernel modules to accelerate some of its work (the DRI).

Alternatives

Userspace: e.g., Xsane



And another alternative is to link a driver with the application that uses it. For example, applications that use scanners are typically linked against `libsane.so`, which provides user-mode device drivers for most USB and SCSI scanners.

User mode drivers?

- More robust
- More functional
- Simpler
- Easier development
- Easier deployment

User mode drivers have a lot of advantages over in-kernel drivers. They are easier to develop, simpler to deploy, less likely to crash the kernel (and more likely to be able to be restarted if the driver itself crashes), can have more functionality than is available in-kernel in a simpler way (no need for user-land helpers)

Easier development

- Use any IDE, any language
 - even Python!
- well ... **Almost no reboots**
- No need to understand Linux in-kernel semantics
 - but *still* have to understand hardware constraints

You don't have to program a user-mode driver in C. You can use a scripted language, to try things out fast (we have an IDE driver written in Python). There's no need to reboot when your driver goes wrong – just `kill -9` the userspace program, and carry on. The interfaces you use are all the standard ones for the language you choose; you don't have to understand all the memory management and blocking arcana for the kernel. And you don't have to change your driver when some part of the kernel changes — for example when preemption was added, or when tasklets were introduced, etc., etc.

Easier deployment

- Kernel ↔ User ABI changes slowly
 - ⇒ Binary distribution of drivers possible
- No licence hassles.
- Use normal package management

User-mode drivers are easier to deploy than in-kernel drivers, once a suitable ABI has been developed. The typical way to deploy an out-of-tree in-kernel driver is to distribute it as source, and to get the installer to build the driver against the current kernel. This requires each machine to have a complete build environment.

User mode drivers can be updated without rebooting. While this can *sometimes* be done now, through judicious use of modules, it's not particularly reliable.

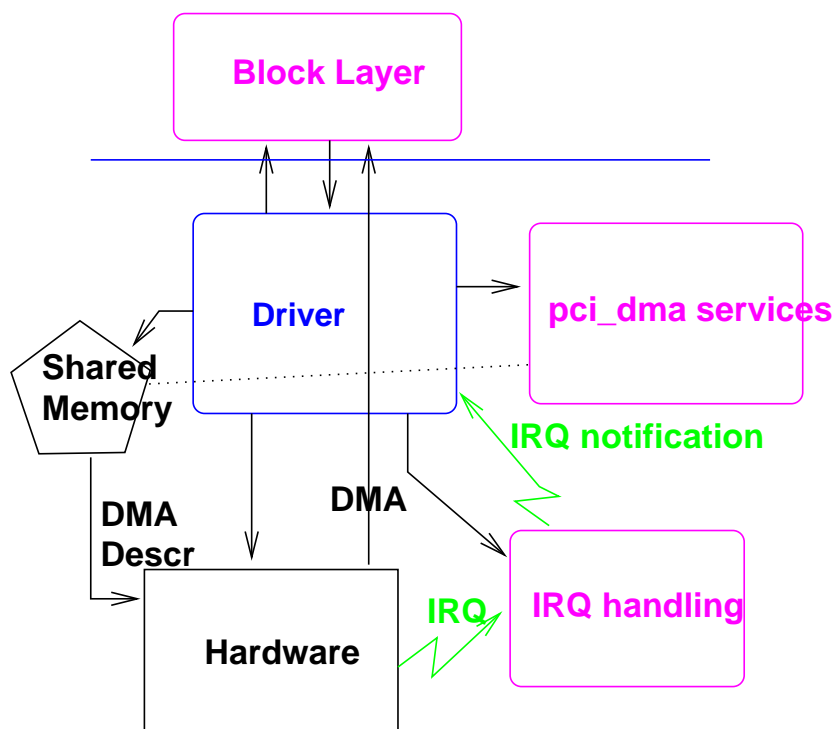
Code that is in any way derived from the Linux kernel code *must* be GPLed. It's unclear to me exactly what 'derived from' means in this context. If you want to release your driver under a different open-source licence (I quite like the MIT licence), you can do it.

Some people may wish to release binary drivers. I don't fully trust closed-source drivers; running them in user space limits the damage

they can do.

Of course, it's always possible for a malicious driver to corrupt things, e.g., by DMAing into random addresses.

Driver Anatomy



What's Needed?

- PCI config space access
- Port/MMIO space access
- **Interrupts**
- **DMA**
- **Zero-copy loop back into kernel**

Linux already provides access to the PCI configuration space via /proc; and by mapping /dev/mem ; libpci.a provides a convenient interface. The first 65526 ports are available via inb() and outb() etc., from user space.

The patch at <http://www.gelato.unsw.edu.au/patches> provides access to interrupts and DMA.

We're currently working on zero-copy loopback.

Interrupts

- Slow — ~3–5 μ s from hardware to acknowledgement
- How to communicate?
 - Signals
 - Event buffer
 - File descriptor

So how are we to communicate interrupt to user space? An interrupt can be modelled as an asynchronous function call, or as a state change. A signal is also an asynchronous function call. A state change could be a wake up on a Futex or similar.

Mapping an interrupt onto a signal at first looks really nice. However, the infrastructure needed to request, mask and unmask interrupts, and to clean up when the interrupt handler dies looks much more like a file descriptor...

Interrupt requirements

- User code to be able to register for an interrupt
- Kernel to unregister
 - on request, or
 - when user process dies
- User code to be informed of interrupt
- User code to be able to mask/unmask interrupt

Looks like file operations...

- `fd = open("/proc/interrupts/58/irq", O_RDONLY);`
 - Allocates interrupt 58; assigns it to current process.
- `x=read(fd, &X, 1)`
 - Unmasks interrupt 58
 - `down(Semaphore)`
 - When interrupt happens, `up(semaphore)`
 - and `read()` returns.

- On `close()`
 - Interrupt masked
 - Interrupt line released for next client

Most user ↔ kernel interfaces nowadays are implemented as filesystem operations, and for good reasons.

Filesystem operations are standard; one doesn't need new system calls.

Filesystems track resources properly, so that dying clients don't leak memory or other things.

Filesystems can report stuff to users easily.

And there's already a filesystem (`/proc`) that exports interrupt information, and allows control — so let's hijack it.

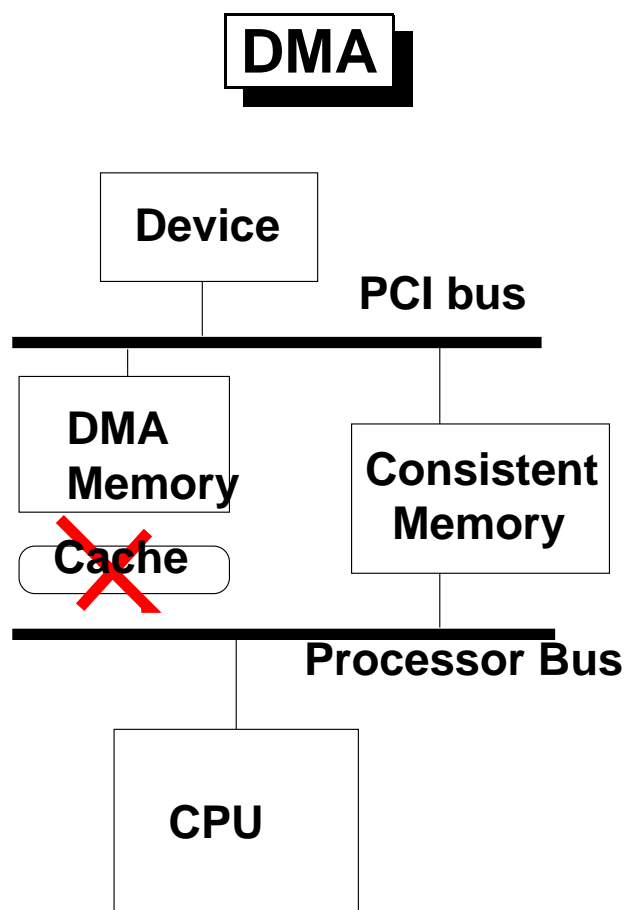
Interrupts: User space

- Typically a separate thread
- Loops:
 - read(irqfd)
 - Acknowledge interrupt to device
 - Queue new work to device (if any)
 - Call client callback

To minimise maximum latency:

- PREEMPTION on
- Real time thread

- Works well
- Latency not measurably increased
→ (despite one extra context switch)

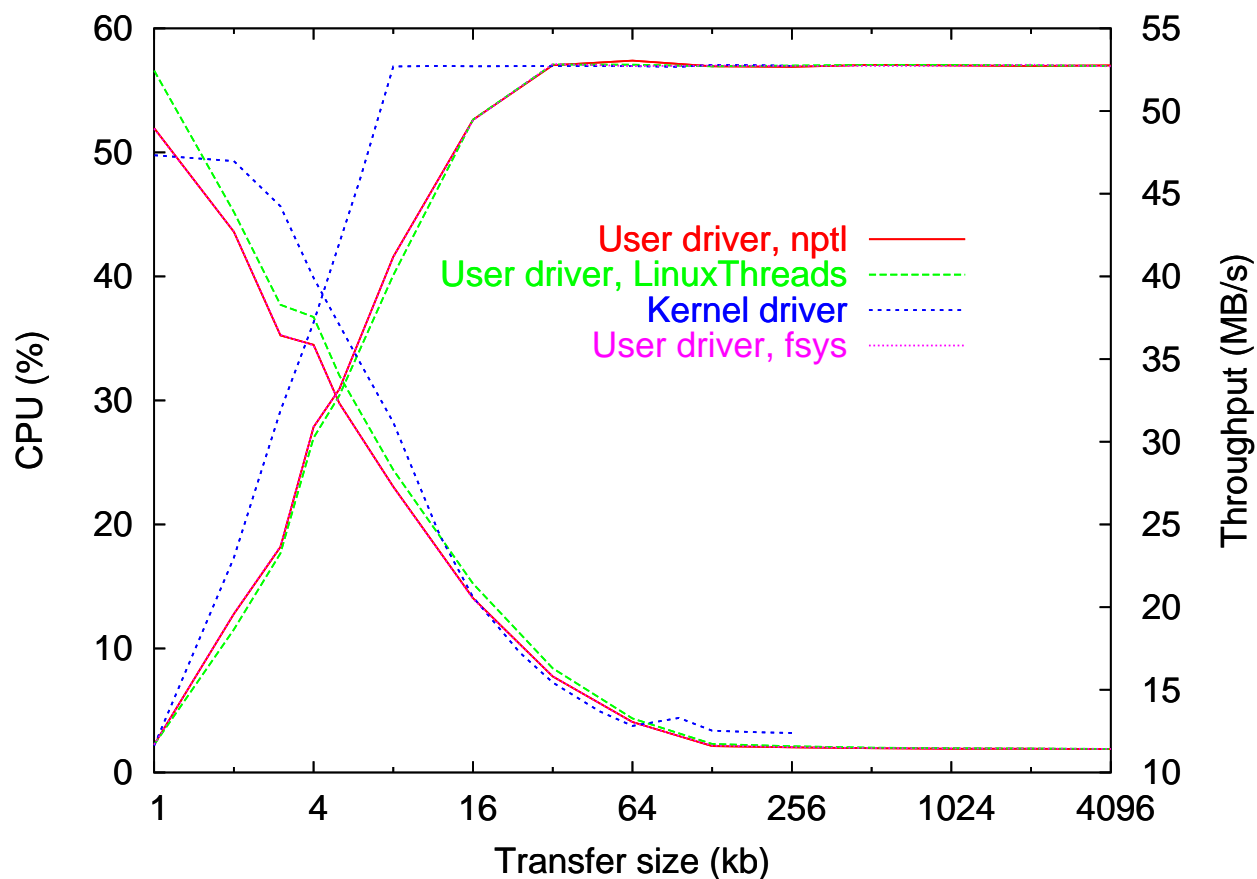


- Unidirectional or Bi-directional
- Bounce Buffers or IOMMU
- Virtually contiguous \nrightarrow physically contiguous
- IOMMU may need PCI device ID

`usr_pci_open(bus, slot, function)`

- Calls `pci_enable_device()` and `pci_set_master()`
- Creates an inode and dentry in the `usrdev` filesystem
- returns file descriptor

Corresponding `close()` operation undoes all this.



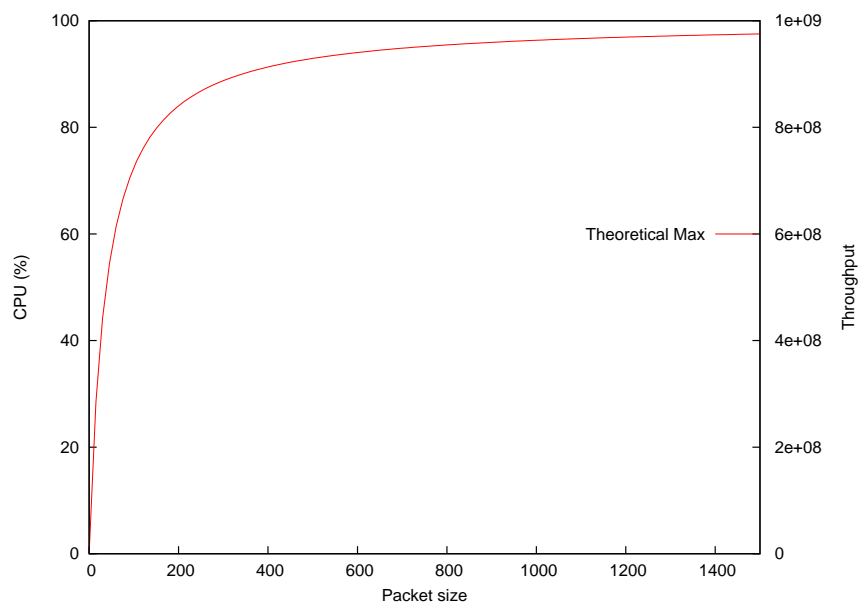
Bonnie results

```

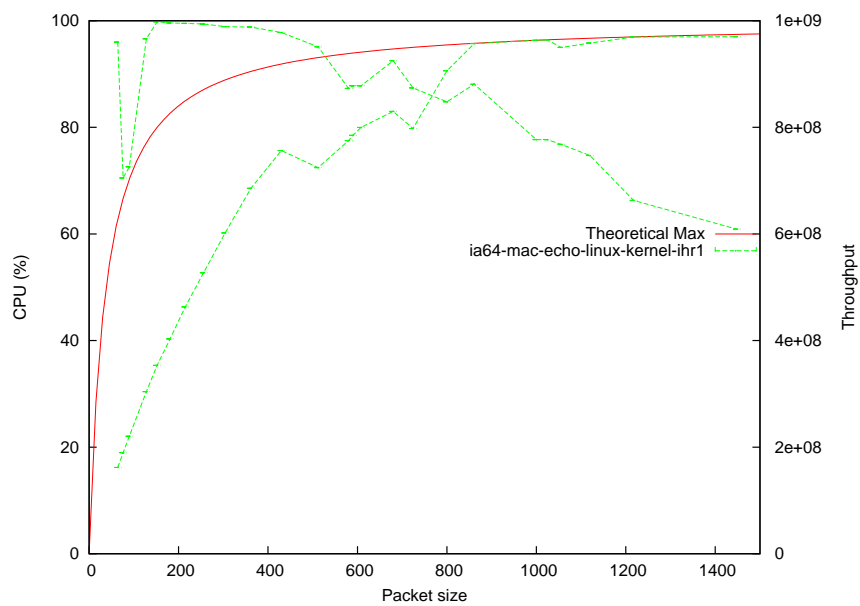
Version 1.03
-----Sequential Output----- --Sequential Input- --Random-
-Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine      Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
kern         1G  6788  99 60439  13 19395   5  6791  97 49659   4 216.2  0
Usr          1G  6788  99 60703  13 21402   6  6765  94 49513   5 194.7  0

-----Sequential Create----- -----Random Create-----
-Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
files:max    /sec %CP /sec %CP /sec %CP sec %CP sec %CP /sec %CP
K 32:1024000:0/100  76  9  81  4  172  0  76  9  32  1  73  0
U 32:1024000:0/100  78  8  81  4  172  0  76  8  31  1  67  0

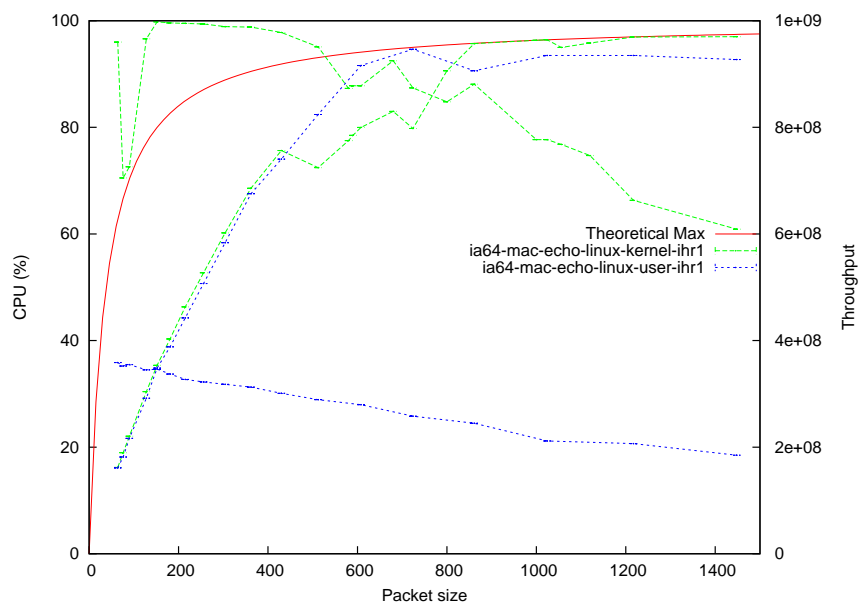
```



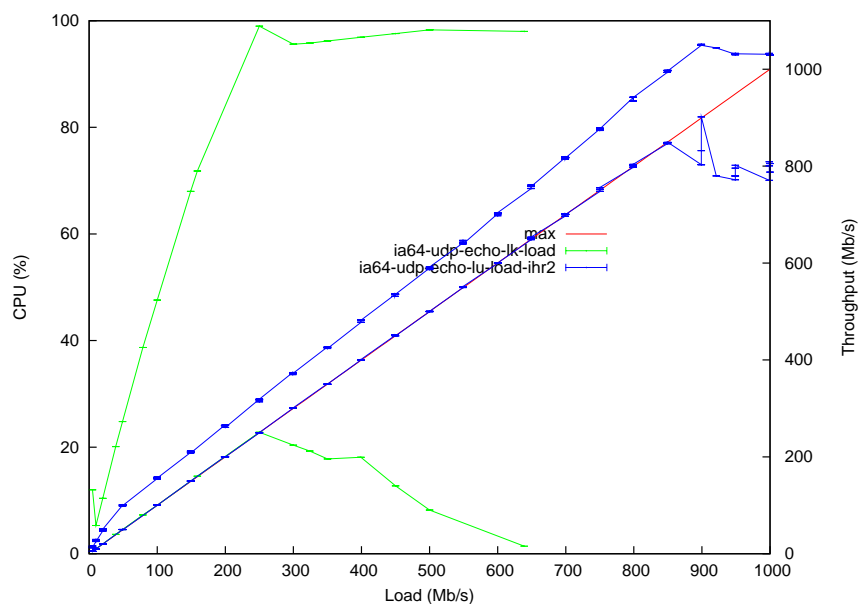
This slide shows the maximum possible bit rates for Gigabit ethernet, depending on packet size, given by the formula $\text{max rate} = \text{packetSize} / (\text{packetSize} + \text{per-packetOverhead})$. The per-packet overhead for gigabit (and other) ethernet is 38 octets, being 8 octets of preamble, 12 octets of address, 2 octet protocol, 4 octet frame-check sequence, and at least 12 octets of inter-packet gap.



The Linux kernel doesn't do a particularly good job at this benchmark.



But user mode, with zero-copy all the way, does do a good job,



For UDP, the situation is worse. The machine becomes totally unresponsive when heavy load is applied. So we turn the benchmark around, and apply gradually increasing load for a fixed (1024 octet) packet size. The user-mode zero-copy driver does very well. The in-kernel and looped-back drivers do very poorly. Why?

Flat profile:

% time	self	cumul	calls	self/call	tot/call	name
14.72	4.36	4.36	-	4.36	-	busyloop@400000000000cba0
8.37	2.48	6.84	2.1271M	1.1659u	1.6007u	kernel:ns83820_do_isr
4.36	1.29	8.13	7.1836M	179.58n	319.95n	kernel:search_extable
3.92	1.16	9.29	21.083M	55.02n	102.3n	kernel:__copy_user
3.61	1.07	10.36	920.46k	1.1625u	3.3739u	kernel:schedule
3.41	1.01	11.37	-	252.5m	-	kernel:dispatch_unaligned_handler
2.73	810m	12.18	676k	1.1982u	5.2u	kernel:do_select
2.50	740m	12.92	1.435M	515.68n	1.8993u	kernel:rx_irq
2.19	650m	13.57	-	216.67m	-	kernel:save_switch_stack
2.19	650m	14.22	1.2395M	524.41n	3.0702u	kernel:rx_refill_atomic
2.16	640m	14.86	7.2577M	88.183n	493.8n	kernel:ia64_handle_unaligned
2.09	620m	15.48	725.6k	854.46n	3.4772u	kernel:ip_rcv

This profile was taken using q-syscollect. The busyloop function is what is measuring idle time, to generate the CPU utilisation curve.

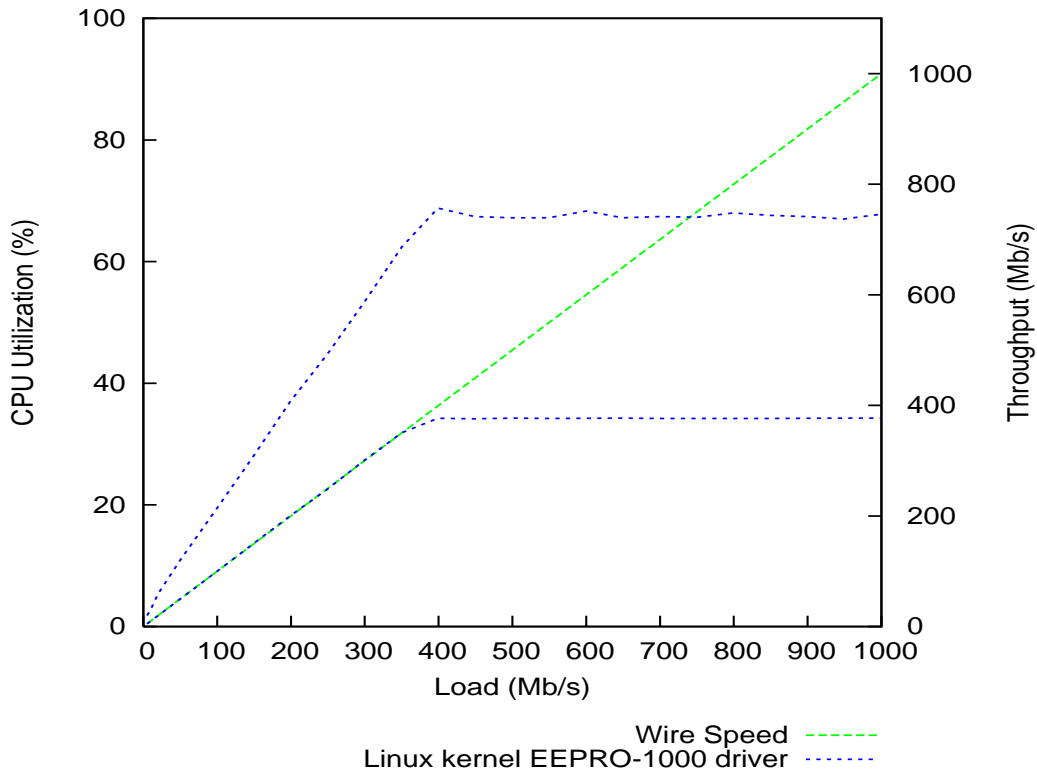
Flat profile:

% time	self	cumul	calls	self/call	tot/call	name
14.72	4.36	4.36	-	4.36	-	busyloop@400000000000cba0
8.37	2.48	6.84	2.1271M	1.1659u	1.6007u	kernel:ns83820_do_isr
4.36	1.29	8.13	7.1836M	179.58n	319.95n	kernel:search_extable
3.92	1.16	9.29	21.083M	55.02n	102.3n	kernel:__copy_user
3.61	1.07	10.36	920.46k	1.1625u	3.3739u	kernel:schedule
3.41	1.01	11.37	4.00	252.5m	1.00	kernel:dispatch_unaligned_handler
2.73	810m	12.18	676k	1.1982u	5.2u	kernel:do_select
2.50	740m	12.92	1.435M	515.68n	1.8993u	kernel:rx_irq
2.19	650m	13.57	-	216.67m	-	kernel:save_switch_stack
2.19	650m	14.22	1.2395M	524.41n	3.0702u	kernel:rx_refill_atomic
2.16	640m	14.86	7.2577M	88.183n	493.8n	kernel:ia64_handle_unaligned
2.09	620m	15.48	725.6k	854.46n	3.4772u	kernel:ip_rcv

The highlighted functions are all to do with the unaligned trap handler inside the kernel. These are being hit because the IHL field in the IP header is declared as a bitfield. ISO C says that bitfields should be part of an int, which is aligned to a 32-bit boundary on IA64. So the compiler generates a 32-bit load to access the field, causing a trap.

For TCP the situation is even worse, as TCP options which come after the IP header are very frequently unaligned.

One might argue that we should use a better device — the DP83820 cannot do DMA other than to 16-byte aligned memory — but unfortunately we see similar results for the EEPRO1000:



Conclusions

- Kernel support for user-mode drivers is possible
- Performance is adequate
- But Need Zero-copy loopback



Gelato

UNSW

Performance and
Scalability on Itanium

www.gelato.unsw.edu.au

Lucy Chubb's Work

Superpages and Page Tables

The Memory Coverage Problem

- Memory is getting bigger
 - Working sets get bigger
 - TLBs stay about the same size
- ⇒ TLB miss overhead grows.

We now measure physical memory in machines in Gigabytes, whereas only a few years ago Megabytes were more convenient. And the main reason for more memory is to use it, with bigger data sets for single programs, and a higher level of multiprogramming.

The TLB

- Fully associative Cache of virtual → physical translations
- Contains GR: page frame number, access rights, whether the page is present, whether it has been accessed etc.
- And ITIR: the page size, and the protection key
- Plus other the Virtual page number and the RID

On Itanium-2 there are two levels of tlbs; a first-level 32-way, and a second level 128-way. Both are fully associative. An L1 miss costs 2 cycles for the Itlb, and 4-cycles for the DTLB. If the L2 TLB misses as well, then the cost is *much* higher. The L1 TLB supports *only* a 4k page size. The L2 TLB supports 4k, 8k, 16k, 64k, 256k, 1M, 4M, 16M, 64M, 256M, 1G and 4G page sizes. Up to 64 of the L2 TLB entries can be pinned in place, so they are not subject to the normal LRU replacement rules.

Some solutions

- Map more per TLB entry
 - Bigger pages
 - Superpages
 - TLB entry sharing between processes

Bigger pages can be used to map more per TLB entry. The default size on IA64 is 16k; on most 32-bit architectures it's 4k. Increasing the page size leads to memory wastage when mapping small objects (still the most common), and increases the latency to swap in a page from disc.

TLB sharing looks like a good idea, but isn't necessarily. Experiments show that it can almost eliminate ITLB misses, but these don't seem to be a bottleneck in general.

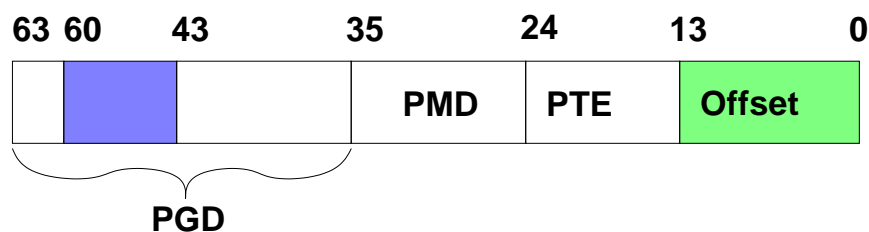
The kind of load that would result in major wins here would be multithreaded code that shared lots of read-mostly data — however, such processes are generally run on multiprocessors, where TLB sharing is not really useful.

Superpage support in Linux

- Some prior work
- Simon Winwood at IBM (now at UNSW)
- Shimizu-san at Tokai University, Japan
- Rohit Seth's work (hugetlbfs — in current kernel)

There are quite a few attempts at superpage (pages bigger than the base page) implementations for Linux. Only the HugetlbFS work is currently in the kernel. The other two main contenders are incomplete, and don't work on IA64 anyway.

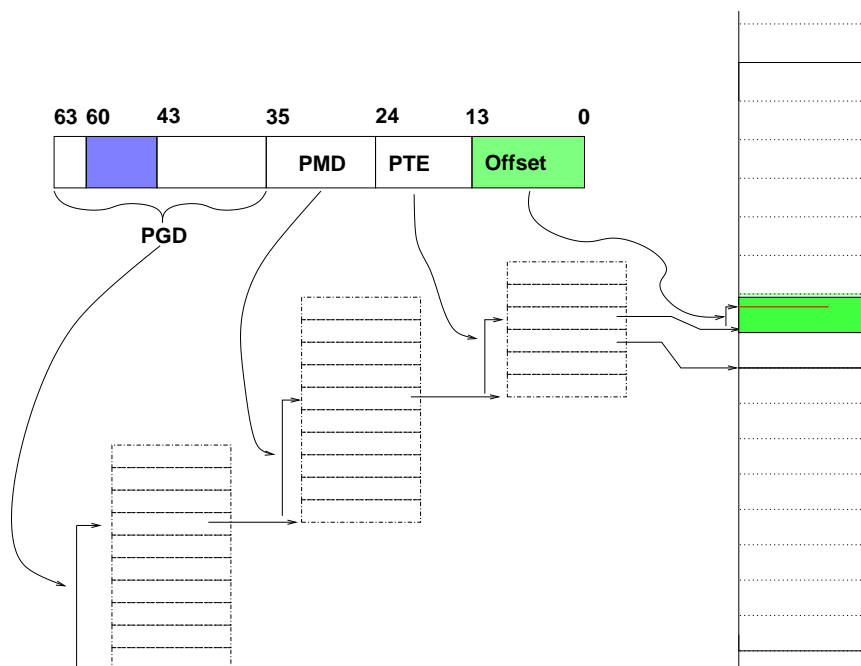
Work at UNSW has started with Shimizu-san's patch and Matt Chapman's Long-format VHPT patch.



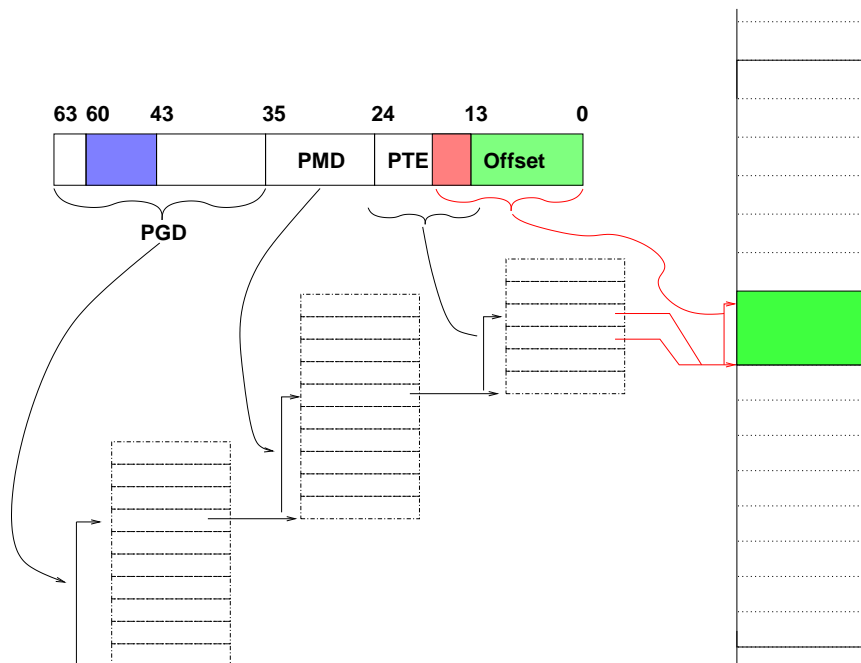
- (Assume 16k pagesize;
`sizeof(void*) == sizeof(pte) == 8`)
- Top three bits are region ID
- RID Concatenated with eight bits from middle of address to give Page Global Directory offset.
- Page Middle Directory is 11 bits
- Page Table index is 11 bits

A virtual address in current Linux is divided into four sections; the top section indexes the top level of the three-level pagetable; then the next and the next; and finally the least significant bits index into the page.

Where you have a 64-bit virtual address, some of the more significant bits are ignored.



This slide shows how the standard Linux pagetable is used to translate a virtual address. From the root of the pagetable (stored in the `struct mm` pointed to by the task structure), the most significant bits index into the global directory giving a pointer to a middle directory page; the next most significant bits then index into that page to get a pointer to the page of Page Table Entries; then the last internal section of the VA is used to index the table of machine-specific PTEs. What's left is used by the architecture-specific code, either directly in the TLB, or as an indication of what to put into the TLB.



In all the current work, where a superpage is involved, the low-level PTEs are replicated, so that when the usual page fault code is exercised it doesn't have to do anything special. However, the PTE it picks up maps not only the page frame decoded by the page fault routine, but also page frames before or after it.

Our work

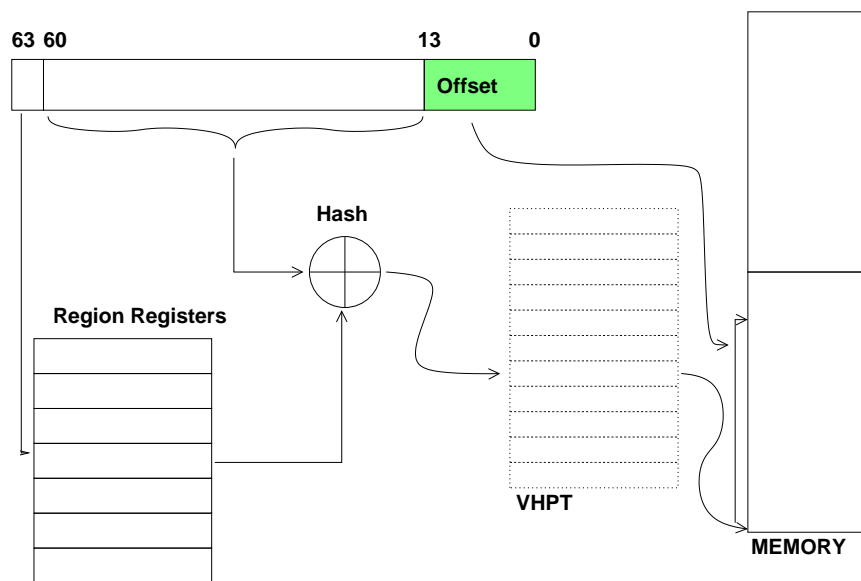
- Based on Shimizu-san's work
- Port in progress to IA64, and to current 2.6 kernel
 - Long-format VHPT.
 - Double-size PTE.
 - * To allow for ITIR
 - * Reduces virtual space available
 - Works on simulator
 - Loops for ever on real hardware

So this is where we're up to as of May 2004.
What follows is future work.

The Page-Table Problem

- 3-level radix tree in current kernel
- Doesn't map well to hardware capabilities.

For our user-mode driver code, under some benchmarks, up to 20% of the time is spent walking the page table to find pages to map into the DMA space. The hardware is capable of much faster access using the long format VHPT.



The long-format VHPT is a hash table, so holds the VPN in each PTE.

The Page-Table Problem

- 3-level radix tree in current kernel
- Doesn't map well to hardware capabilities.
- Each architecture walks the actual pagetable data structures
 - no clean interface.
 - VHPT is cache of Linux page table
- Hard to extend for superpages
 - Replicate PTEs?
 - How to get more information higher up the tree?

We're currently trying to define a clean interface to the page table, then reimplement the three-level page table in terms of this interface, then start implementing new page table structures.

So ...

- Clean page table interface
- Replaceable pagetable implementations
 - Including the existing one!
 - And the GPT

See [IA64 Wiki: PageTableInterface](#) for more details.

Goals

- Mapped object (VMA) Sharing
- Superpage support
- Performance!



References

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. (2001). An empirical study of operating systems errors. In *Symposium on Operating Systems Principles*, pages 73–88. <http://citeseer.nj.nec.com/article/chou01empirical.html>.

Swift, M., Martin, S., Leyland, H. M., and Eggers, S. J. (2002). Nooks: an architecture for reliable device drivers. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France.