Initial Evaluation of a User-Level Device Driver Framework

Kevin Elphinstone[1] and Stefan Götz[2]

[1] National ICT Australia***
and
School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
kevine@cse.unsw.edu.au

[2] System Architecture Group, Universität Karlsruhe, 76128 Karlsruhe, Germany
sgoetz@ira.uka.de

Abstract. Device drivers are a significant source of system instability. In this paper, we make the case for running device drivers at user-level to improve robustness and resource management. We present a framework for running drivers at user-level whose goal is to provide similar performance when compared to in-kernel drivers. We also present initial promising performance results for the framework.

1 Introduction

Most modern operating systems feature monolithic operating system kernels. Most modern architectures are designed to efficiently support this form of construction. A kernel provides its services by combining the software that implements potentially independent services into a single large amalgamation. However, once we scale the size and complexity of a monolithic system to the levels of current systems, extensibility becomes more difficult due to legacy structure, security becomes more difficult to maintain and impossible to prove, and stability and robustness also suffer.

One promising approach to tackling the expanding complexity of modern operating systems is the microkernel approach [1]. A microkernel-based OS consists of a very small kernel at its core. The kernel only contains a minimal set of services that are efficient and flexible enough to construct services for applications as servers running on the microkernel. Only the microkernel itself runs in privileged mode. Although these servers provide operating system functionality, they are regular applications from the microkernel's point of view. Such a system enables extensibility as servers can be added or removed, it provides security as the core of the system is small enough to analyse or maybe even prove [2], and

*** National ICT Australia is funded by the Australia Government's Department of Communications, Information and Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 256–269, 2004.
© Springer-Verlag Berlin Heidelberg 2004

stability and robustness is improved as services can be isolated from each other. The modular structure that is encouraged, and even enforced by virtual memory protection boundaries, improves maintainability.

Most microkernel based systems still include device drivers in the kernel. Drivers are included for either security [3], performance reasons [4] , or because the system's focus was toward goals other than decomposition and minimisation, such as distribution [5,6,7]. It has been shown that device drivers exhibit much higher bug rates (three to seven time higher) than other kernel code [8]. Microsoft has also identified drivers as the major cause of system instability and has instigated their driver signing program to combat the problem [9]. It remains to be seen whether signing a driver as having passed a quality control scheme has an affect on driver correctness. Simply digitally signing a piece of software obviously has no effect on the software itself.

This paper tackles the problem of device driver instability by running drivers at user-level and hence subjecting them to the normal controls applied to applications. We also aim to provide a flexible driver framework for microkernel-based systems that enables trade-offs between driver performance and containment. Attempts thus far can be characterised as being too concerned with compatibility with existing driver collections [10] or having an alternative focus such as realtime systems [11]. The achieved performance has been insufficient to make the approach convincing.

Device drivers at user level could be treated almost like normal applications. Like normal applications, drivers could be isolated from unneeded resources using the processor's virtual memory hardware. Being able to apply the principle of *least privilege* would greatly minimise the potential damage a malfunctioning driver could inflict. This is very much in contrast to the current situation where drivers have access to all resources in the system. A single malfunction often results in catastrophic failure of the entire system.

Developers of user-level drivers can use facilities usually only available to normal applications. Standard debuggers can provide a much richer debugging environment than usually available to kernel-level drivers (e.g., source level debugging versus kernel dumps). Application tracing facilities can also be used to monitor driver behaviour. Application resource management, such as CPU time controls, can be used to control driver resource usage.

User-level drivers are not a completely new idea. Drivers in the past have been incorporated into applications such as networking software (e.g distributed shared memory applications) [8,12]. The inclusion in this case was to improve performance by giving the application direct access to the device, and thus avoiding kernel entry and exit. Such scenarios relied on near exclusive access to the device in order to avoid issues in multiplexing the device between competing clients. In most cases, specialised hardware was developed to provide concurrent access via specialised access channels, and to provide performance via a specialised interface that required no kernel intervention.

We propose an architecture where the system designer can choose the most appropriate configuration for drivers based on requirements of the targeted sys-

tem. We envisage drivers incorporated into specialised applications where performance is paramount. However, we also envisage drivers running as individual servers to improve security and robustness, or drivers clustered into a single server to reduce resource requirements. Immature drivers could be run in isolation until mature enough to be combined with other components when required.

While we intend to take advantage of specialised hardware (such as myrinet network cards which have their own programmable processors [13]), we also do not intend to restrict ourselves to such hardware. For the results of this project to be truly useful we must be able to support commodity hardware that is not necessarily tailored to the environment we are developing. Commodity hardware may not provide all features necessary for complete security. For instance, nearly all hardware is unable to restrict what a driver can access via DMA. On such a hardware platform, a malicious driver can always corrupt a system. However, even limited success in supporting commodity hardware with little performance impact would make our results applicable to the widest variety of platforms possible. Limited success could persuade more manufacturers to include the hardware features required for complete security. Our group has begun exploring restricting DMA access using the limited hardware available in high-end servers [14]. However, we do not focus on this problem for the remainder of this paper.

Past approaches to drivers at user-level have usually taken a top-down approach. The system was designed with a specific target in mind, built, and analysed. The results have varied widely. Some projects, specifically the user-level networking with specialised hardware, have been successful [12,15]. Other projects have been less successful and have usually disappeared without a clear analysis of why success eluded them [10,16]. In this paper we identify the fundamental operations performed by device drivers, their relevance to performance, and present how they can be implemented safely and efficiently at user-level.

In the remainder of the paper, Section 2 provides the background to running user-level drivers by describing a simple model of device drivers in existing monolithic systems. We use it as a reference for the rest of the paper. Section 3 describes the experimental operating system platform upon which we developed our driver framework. Section 4 describes the framework itself. The experimental evaluation and results follow in Section 5, with conclusions afterwards in Section 6.

2 Simple Driver Model

To define common terminology, help convey the issues we have identified, and introduce our framework itself, we present a simple model of a device driver and highlight the issues within that model. This model initially assumes a traditional monolithic kernel whose kernel address space is shared between all process contexts.

A driver broadly consists of two active software components, the *Interrupt Service Routine* (ISR) and the *Deferred Processing Component* (DPC). We ignore initialisation code and so forth. The ISR is responsible for reacting quickly

and efficiently to device events. It is invoked almost directly via a hardware defined exception mechanism that interrupts the current flow of execution and enables the potential return to that flow after completion of the ISR. In general, the length of the ISR should be minimised so as to maximise the burst rate of device events that can be achieved, and to reduce ISR invocation latency of all ISRs (assuming they are mutually exclusive).

The ISR usually arranges for a DPC to continue the processing required to handle the device event. For example, a DPC might be an IP stack for a network device. A DPC could also be extra processing required to manage the device itself, or processing required to complete execution of a blocked kernel activity. Another way to view a DPC is that it is the kernel activities made runnable as a result of the execution of the ISR. It may be a new activity, or a previously suspended activity. DPCs are usually activated via some kernel synchronisation primitive which makes the activity runnable and adds it to the scheduler's run queue.

2.1 Driver Interfaces and Structure

A driver consists of an interface in order for clients (other components in the kernel) to direct the driver to perform work. For instance, sending packets on a network device. Drivers also expect an interface provided by the surrounding kernel in order to allocate memory, activate DPCs, translate virtual addresses, access the device information on the PCI bus, etc. We believe the following interfaces are important to driver performance:

Providing work to the driver. Drivers provide an interface for clients to enqueue work to be performed by the device. This involves passing the driver a work descriptor that describes the work to be performed. The descriptor may be a data structure or arguments to a function call. The work descriptor identifies the operation and any data (buffers) required to perform the work. Drivers and clients share the kernel address space which enables fast transfer (by reference) and access to descriptors and buffers.

DPCs and offloading work. Drivers also produce work for clients. A common example is a network driver receiving packets and therefore generating work for an IP stack. Like enqueueing work for the driver itself, an efficient mechanism is required for the reverse direction to enqueue work for, and activate, a DPC such as an IP stack. Work descriptors and buffers can be handled in a similar manner to enqueueing work for the driver, i.e. descriptors and buffers can be transferred and accessed directly in the kernel's address space.

Once work is enqueued for a DPC, the DPC requires activation via a synchronisation primitive. Again, the primitive can rely on the shared kernel address space to mark a DPC runnable and place it on the appropriate scheduler queue.

Buffer allocation. The buffers containing the data that is provided to the driver must be allocated prior to use and deallocated for reuse after processing. Buffers may be produced by a client and consumed by a driver (or vice

versa) and are managed via a memory allocator (e.g. a slab allocator) in the shared kernel address space.

Translation. Buffers specified by user-level applications are identified using virtual addresses. DMA-capable devices require these addresses to be translated into a physical representation. This translation can be done simply and quickly by the device driver by accessing the page tables stored within the kernel address space. Additionally, some driver clients deal only with the kernel address space and can use physical addresses directly (or some fixed offset).

Pinning. DMA-capable devices access physical memory directly without any mediation via a MMU (though some architectures do possess I/O MMUs). Coordination between the page replacement policy and the device driver is required to avoid the situation where a page is swapped out and the underlying frame is recycled for another purpose while an outstanding DMA is yet to complete. Preventing pages from being swapped out is generally termed *pinning* the page in memory. This can be implemented with a bit in the frame table indicating to the page replacement algorithm that the frame is pinned.

Validation. Validation is the process of determining whether a request to the driver is permitted based on knowledge of the identity of the requester and the parameters supplied. Validation is simple when a client issues a request to a driver in a shared address-space kernel. The driver can implicitely trust the client to issue sensible requests. It only needs to check the validity of a request for robustness reasons or debugging. If needed, the client module in the kernel is usually responsible for the validity of any user-level supplied buffers or data which needs to reside in memory accessible to the user-level application. Such a validation is simple and inexpensive to perform within a shared address-space kernel — all the data required to validate an application's request is readily available.

It is clear that the model envisaged by computer architects is a fast hardware-supported mechanism to allow privileged drivers to respond to device events, and that the drivers themselves have cheap access to all the information required to perform their function via the privileged address space they share with the kernel. The high degree of integration with the privileged kernel allows drivers to maximise performance by minimising overheads needed to interact with their surrounds. This high degree of integration is also the problem: drivers detrimentally affect security, robustness, and reliability of the entire system.

3 Experimental Platform

We chose the L4 microkernel as the experimental platform for developing and evaluating our driver framework[1]. L4 is a minimal kernel running in privileged mode. It has two major abstractions: threads and address spaces. Threads are the unit of execution and are associated with an address space. A group of threads

within an address space forms a task. Threads interact via a very light weight synchronous interprocess communication mechanism (IPC) [17].

L4 itself only provides primitive mechanisms to manage address spaces. Higher-level abstractions are needed to create a programming environment for application developers. The environment we use is a re-implementation of a subset of the SawMill multi-server operating system developed for L4 at IBM [10], called *Prime*. The most relevant component to this paper is the virtual memory framework [18], which we will briefly introduce here.

Dataspaces are the fundamental abstraction within the VM framework. A dataspace is a container that abstracts memory objects such as files, shared memory regions, frame buffers, etc. Any memory that is mappable or can be made mappable can be contained by a dataspace. For a thread to access the data contained in a dataspace, the dataspace is *attached* to, i.e. mapped into, the address space. Address spaces are constructed by attaching dataspaces including application code and data, heap and stack memory.

Dataspaces themselves are implemented by *Dataspace Managers*. Any task within the VM framework can be a dataspace manager by implementing the dataspace protocol. For example, a file system dataspace manager provides files as attachable dataspaces by caching disk contents within its address space, and using the underlying L4 mechanisms to map the cached content to clients who have the dataspace attached. Dataspace managers map pages of dataspaces to clients in response to the page fault handling mechanism which forwards page faults on attached dataspaces to the appropriate dataspace manager that implements the dataspace.

The dataspace and dataspace manager paradigms provide a flexible framework of object containers and object container implementors. Few restrictions are placed on participants other than implementing the defined interaction protocol correctly. However, while clients with attached dataspaces see a logical container, device drivers interacting with such a container require more information about the current dataspace state for DMA purposes. In particular, they have to know the translation between dataspace addresses and physical memory which is only known by the dataspaces' manager. In a traditional system we have the kernel implemented page tables as a central authority for translation information. With our VM framework, translation information is distributed amongst dataspace managers which creates the problem of efficient information retrieval. We describe our solution to this problem in Section 4

4 Driver Framework

As described in Section 2, the high degree of hardware and software integration in classic system architectures creates an environment for efficient driver implementation. The challenge is to keep the high level of integration when transforming drivers into user-level applications while enforcing protection boundaries between them and the surrounding system. There are obviously trade-offs to be made between the strength of the protection boundary and the cost of interacting across

it. A network driver interface that copies packets across protection boundaries provides greater packet integrity and poorer performance compared to an interface that passes packets by reference. In choosing trade-offs for this paper we focused on maximising performance while still improving robustness. Drivers and their clients may corrupt the data they produce and consume, but should not be able to corrupt the operation of each other. However, our framework is not restricted to the particular trade-offs we made for this paper. A system designer can increase or decrease the degree of isolation between clients and drivers by small changes in interfaces, their implementation, or the composition of drivers and clients.

For this paper we took the following approach:

- Minimise the cost of interaction between clients and drivers by interacting via shared memory instead of direct invocation where possible. This sharing is secure in that it is done such that clients cannot interfere with the operation of drivers and vice versa. However, data buffers can be modified by clients or drivers at any point in the interaction.
- Minimise the cost of any overhead we must insert between clients and drivers (or between drivers and the kernel) to support interaction across protection boundaries.
- For any overhead that we must insert to enable interaction, we attempt to amortise the cost by combining operations or event handling where possible.

We now describe how we applied our approach to constructing a driver framework with reference to the model introduced in Section 2.

4.1 Interrupts

Direct delivery of interrupts to applications is not possible on current hardware. A mechanism is required for an ISR within a driver application to be invoked. We use the existing model developed for L4 where interrupts are represented as IPCs from virtual *interrupt threads* which uniquely identify the interrupt source. The real ISR within the kernel masks the interrupt, transforms the interrupt event into an IPC message from the interrupt thread which is delivered to the application's ISR. The blocked ISR within the application receives the message, unblocks, and performs the normal ISR functionality. Upon completion, the driver ISR sends a reply message to the interrupt thread resulting in the interrupt source being unmasked. The ISR can then block waiting for the next interrupt IPC.

While L4 IPC is very light-weight, it is not "free". We add a small amount of direct overhead to implement this clean model of interrupt delivery. Indirect overhead is incurred by context switching from an existing application to the driver application upon interrupt delivery. We expect this overhead to be low compared to the high cost of going off-chip to manage devices, and plan to reduce the overall overhead by using interrupt hold-off techniques currently applied to limit the rate at which interrupts are generated.

4.2 Session-Based Interaction

Copying data across protection boundaries is expensive. Where possible, we use shared memory to pass data by reference, or to make control and metadata information readily available to clients and drivers. Establishing shared memory is also an expensive operation both in terms of managing the hardware (manipulating page tables and TLB entries), and in terms of performing the book-keeping required in software. To amortise the cost of setting up shared memory we use a session-based model of interaction with drivers.

A *session* is the surrounding concept within which a sequence of interactions between client and driver are performed. It is expected that a session is relatively long lived compared to the duration of the individual interactions of which we expect many within a session. To enable pass-by-reference data delivery, one or more dataspaces can be associated with a session for its duration. Dataspaces can contain a shared memory region used to allocate buffers, a client's entire address space, or a small page-sized object. There are obviously trade-offs that can be made between cost of establishing a session, and the size and number of dataspaces associated with a session. To avoid potential misunderstanding, there can be many underlying sessions within our concept of a *session*. For example, an IP stack has a *session* with the network device driver through which many TCP/IP sessions can be managed.

4.3 Lock-Free Data Structures

There are obvious concurrency issues in managing data structures in shared memory. We make heavy use of lock-free techniques to manage data structures shared between drivers and their clients. We use lock-free techniques for predominately two reasons: to avoid external interaction and to avoid time-outs and recovery on locks.

Enqueueing work (packet/command descriptors and similar metadata) for a driver by explicitly invoking it requires at least two context switches per enqueued item. This would cause the high level of integration achieved in normal systems to be lost. Lock-free queues (implemented with linked lists or circular buffers) allow work to be enqueued for a driver (or a client) without requiring explicit interaction with the driver on every operation. This encourages a batching effect where several local lock-free operations follow each other, and finally the recipient driver is notified via explicit interaction (a queued-work notify event).

Lock-free techniques allow us to avoid dealing with excessive lock holding times. It is much easier to validate potentially corrupt data in a lock-free queue that is caused by a misbehaving client (we have to validate client provided data anyway), than to determine if a client is misbehaving because a lock is found held.

4.4 Translation, Validation, and Pinning

Drivers process work descriptors which can contain references to the actual buffers to be processed. Buffers are specified as ranges of addresses within datas-

paces. The dataspaces are associated with the surrounding driver-client session. The dataspaces themselves are implemented by other applications (dataspace managers). This creates an interesting problem. The knowledge of a dataspace's existence, who is accessing it, and what physical frames implement it at any instant in time is known by the dataspace manager implementing the dataspace, not the client using the dataspace, and not the driver accessing the dataspace to process the requests of the client. In a traditional system, this information (page tables and frame tables) is readily available to the driver within the kernel address space. Ideally, we would again like to safely replicate the high degree of integration between driver, clients, and information required to operate.

The validation of buffers specified by the client within the above framework is simple. Given buffers are ranges of addresses within dataspaces, validation is a matter of confirming the dataspace specified is associated with the session between the driver and client.

The translation of dataspace pages to physical frames is required by drivers of DMA-capable devices. This translation is only known by a dataspace manager. Our approach thus far has been to avoid external interaction by the driver as much as possible, however translation requires this interaction in some form. To enable translation, the dataspace manager provides a shared memory region between it and the device driver: the translation cache. The translation cache is established between the manager and driver when a dataspace is added to a session between the driver and client. Multiple dataspaces from the same manager can share the same translation cache. The translation cache contains entries that translate pages within dataspaces into frames[1]. The cache is consulted directly by the driver to translate buffer addresses it has within dataspaces to physical addresses for DMA. After the translation cache is set up, the driver only needs to interact with the object implementor in the case of a cache miss. At present we use a simple on-demand cache refill policy, but we plan to explore more complex policies if later warranted.

In addition to translating a buffer address to a physical address for DMA, the driver needs a guarantee for the duration of DMA that the translation remains valid, i.e. the page (and associated translation) must remain *pinned* in memory. In this paper we have not focused on the problem of pinning in depth. We see at least two approaches to managing pinning for DMA. The first method is to use time-based pinning where entries in the translation cache have expiry times. The second method is to share state between the driver and dataspace implementor to indicate the page is in use and should not be paged out.

Time-based pinning has the difficult problem of the driver needing to estimate how long a DMA transaction might take, or even worse, how long it will take for a descriptor in a buffer ring to be processed, e.g. on a network card. However, time-based pinning has the nice property of not requiring interaction between

[1] In our virtual memory framework, dataspaces can also be composed of other dataspaces. In this case, the translation consist of a sequence of dataspace to dataspace translations, and then a final dataspace to physical frame translation. However, we ignore this scenario for the sake of clarity in the paper.

driver and object implementor. Further discussion of time-based pinning can be found our previous work [19].

State sharing to indicate to the dataspace manager that pinning is required could be achieved with a pin-bit within translation cache entries. This requires read-write shared memory between driver and dataspace implementor that was not required up until this point. It should be clear that the pin-bit has direct parallels with similar flags in a traditional frame table and thus warrants little further discussion. Note that the pin-bit would only be advisory. The memory implementor can enforce quotas on pin time or the amount of pinned memory by disabling the driver and resetting the device (if permitted) to recover pinned pages.

4.5 Notification

Unlike traditional systems where thread state and scheduler queues are readily available in shared kernel space, in a system with drivers in separate protection domains, system calls must be performed to manipulate the scheduler queues, i.e. block and activate threads. System calls are significantly more expensive than state changes and queue manipulations. An efficient activation mechanism is required for ISRs to hand-off work to DPCs, and for both clients and drivers to deliver work and potentially block as the sender and while activating the recipient.

By using queues in shared memory for message delivery, we create the environment required for user-level IPC (as opposed to IPC involving the kernel). User-level IPC has been explored by others [20,21], mostly in the context of multiprocessors where there is an opportunity to communicate without kernel interaction via shared memory between individual processors. Our motivation is two-fold. We wish to avoid kernel interaction (not activate the destination) if we know the destination is active (or will become active), and we wish to enable batching of requests between drivers and clients by delaying notifications when possible and desirable.

Our notification mechanism is layered over L4 IPC. Blocking involves waiting for a message, activating involves sending a message. To avoid notifications when unnecessary, the recipient of notifications indicates its thread state via shared memory. If marked inactive, a notification is sent; if not it is assumed that the recipient is (or will be) active and the notification is suppressed.

The delay between setting the state and blocking waiting for IPC creates a race condition if preempted between the modification and blocking waiting for IPC. There is a potential for notification messages to be missed if sent to a thread that has not yet blocked. However, if the sender does not trust the recipient, it is not safe for the sender to block on or re-send notifications without being vulnerable to denial-of-service attacks. Thus, recipients have to be able to recover from missed notifications on their own. We resolve this race by using a general mechanism called *preemption control*, which can make threads aware of preemption. In the rare case that a preemption is detected, the recipient rolls back to a safe active state from where it tries to block again.

The notification bit creates opportunities for delaying notification (to increase batching) or avoiding notification altogether. An example of avoidance is where a network driver would eventually receive a “packet sent” or “transmit queue empty” interrupt from the device. If such events are known to occur within acceptable latency bounds, notifying such a device when enqueueing an outgoing packet is unnecessary as the driver will eventually wake via the interrupt to discover the newly enqueued packets. This allows a driver client to submit requests continuously to maximise the batching effect.

5 Evaluation and Results

We evaluate our framework for running device drivers at user level in a network context. Handling modern high-speed networks is challenging for traditionally structured systems due to the very high packet rate and throughput they achieve.

Our test system consists of a user-level ISR that is comprised of generic low-level interrupt handling in the L4 kernel and device-specific interrupt handler for a dp83820 Gigabit ethernet card driver. The DPC is the lwIP IP stack and a UDP echo service that simply copies incoming packets once and echoes them to the sender. The driver and lwIP execute in separate processes which interact as described in Section 4. Note that the echo service is compiled into the process containing lwIP. The machine is a Pentium Xeon 2.66 GHz, with a 64-bit PCI bus.

We chose this test scenario as we believe it to be the extreme case that will expose the overheads of our framework most readily. The test does very little work other than handle interrupts, and send/receive packets across a protection boundary between the driver and lwIP, and then onto (or from) the network. In a more realistic scenario, we would expect the “real” application to dominate CPU execution compared to the drivers and IP stack. By removing the application, the driver and lwIP stack (and our overheads) will feature more prominently.

We used the ipbench network benchmarking suite[22] on four P4-class machines to generate the request UDP load that we applied to the test system. ipbench can apply specific load levels to the target machine, the packet size used was 1024 bytes. We performed two experiments, (a) that uses random-interval program counter sampling to develop an execution profile at an offered load of 450Mb/s using 100us interrupt hold-off, and (b) which ramps up the offered load gradually and records throughput and CPU utilisation at each offered load level. The load generators record echoed packets to calculate achieved throughput, CPU utilisation is measured by using the cycle counter to record time spent in a low priority background loop. Utilisation results obtained via random sampling and the cycle counter agree within one percent. The second experiment was performed for both 0μs and 100μs interrupt hold-off for Prime, and to compare, Linux with our driver and Linux’s IP stack in-kernel, and user-level echo server.

The results show that for the profile experiment 68% of time was spent in the idle thread. For the remaining 32% of samples, we divided the samples into the following categories: *IPC* kernel code associated with the microkernel IPC

path; *Driver* code associated with the network driver that would be common to all drivers for this card independent of whether it runs in-kernel or at user-level; *IP* code associated with lwIP that is also independent of running at user-level or in kernel; *Kernel* code that is independent of system structuring, this code is mostly related to interrupt masking and acknowledgement; User-level *Notification* code that implements notification within our framework; User-level *Translation* code that performs translation from dataspace addresses to physical memory; User-level *Interrupt* code related to interrupt acknowledgement; User-level *Buffer* code for managing packet buffers, including (de-)allocation, within our framework.

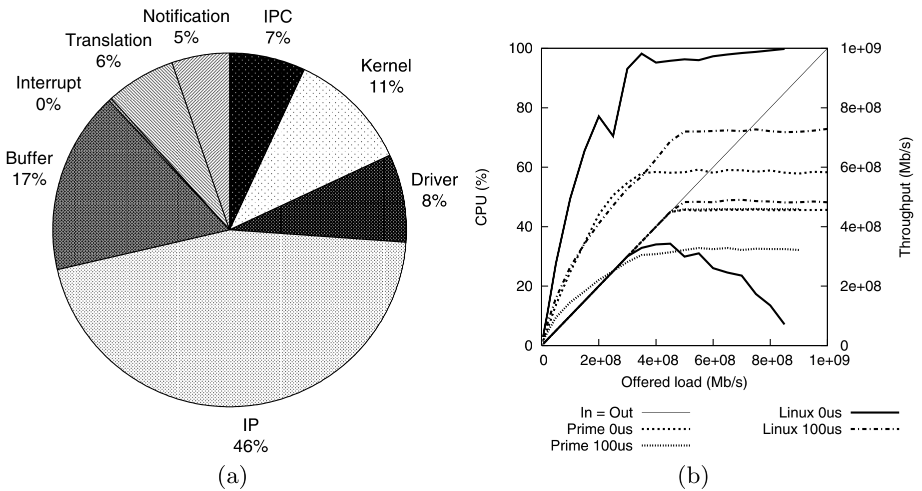

Fig. 1. (a) Execution profile. (b) CPU utilisation and throughput versus load level for Prime and Linux using 0μs and 100μs interrupt hold off.

The profile of execution within these categories is illustrated in Figure 1. The component of execution unrelated to our framework (Kernel + Driver + IP) forms 65% of non-idle execution time. Code related to our framework (Buffer + Interrupt + Translation + Notification + IPC) forms the remaining 35% of non-idle execution. Even when considering all framework related code as overhead introduced by running drivers at user-level, this is not a bad result. The test scenario we chose to analyse does so little work that we expect in a more realistic scenario our framework will consume a smaller fraction of execution time.

Considering all framework related code as overhead is not a fair comparison as two components of the framework (buffer management and translation) also have to be performed in a traditional system structure. If traditional buffer management and translation is comparable, then the overhead of running drivers at user-level (Notification + IPC + Interrupt) is only 12% of non-idle time.

Figure 1 also shows the result of the CPU utilisation and throughput experiment. The thin diagonal line represents where achieved throughput equals offered load. The lines beginning and rising above this reference represent CPU utilisation. The lines that track the reference and diverge to the right represent achieved throughput. We see that for 100μs interrupt hold off, both Prime and Linux achieve similar throughput of approximately 460Mb/s and 480Mb/s respectively. Prime uses much less CPU achieving the result (32% versus 72%). However, we make no claim of a fair comparison as Linux has a heavier weight IP stack, translation and pinning infrastructure, and uses a socket interface which results in an extra packet copy compared to Prime. We simply observe that we are currently competitive with a traditionally-structured existing system and are optimistic we can at least retain comparable performance in more similarly structured systems. For the 0μs hold-off results, we see Linux goes into livelock near 100% CPU after which throughput tapers off as offered load increases. Prime achieves exactly the same throughput for 0μs and 100μs hold-off, though CPU utilisation differs markedly (58% versus 32%).

6 Conclusions

We have constructed a framework for running device drivers at user-level. Our goal was to preserve the high degree of system integration that enables high-performance driver construction while at the same time confining drivers safely to their own address space like normal applications. We analysed our framework's performance in the context of gigabit ethernet, and our initial results show modest overhead in an execution profile in a test scenario designed to exacerbate the overhead. In throughput oriented benchmarks, we demonstrated similar performance to Linux in terms of achieved throughput. We plan to further explore our framework's performance by constructing more realistic test scenarios (e.g. SPECweb), drivers and interfaces for other devices (e.g. disk). We also plan to explore system structures more comparable to existing systems (e.g. driver, IP stack, and web server all running as separate processes).

We eventually hope that our results will be encouraging enough to CPU and system architects to consider exploring efficient control of DMA for protection purposes in commodity hardware. Such hardware would ensure that device drivers are just normal applications under the complete control of the operating system.

References

1. Liedtke, J.: Toward real microkernels. Communications of the ACM **39** (1996)
2. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel - the VFiasco project. In: Proc. 10th SIGOPS European Workshop. (2002)
3. Engler, D.R., Kaashoek, M.F., Jr., J.O.: Exokernel: An operating system architecture for application-level resource management. In: 15th Symp. on Operating Systems Principles, Copper Mountain Resort, CO, ACM (1995)

4. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M.: MACH: A new kernel foundation for UNIX development. In: Proc. Summer USENIX. (1986)
5. Cheriton, D.R.: The V kernel: A software based for distribution. IEEE Software **1** (1984) 19–42
6. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Hermann, F., Kaiser, C., Langlois, S., Leonard, P., Neuhauser, W.: Chorus distributed operating system. Computer Systems **1** (1988)
7. Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G.J., Mullender, S.J.: Experiences with the amoeba distributed operating system. Communications of the ACM **33** (1990) 46–63
8. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. (In: Proc. 18th Symp. on Operating Systems Principles)
9. Microsoft: Driver signing for windows. Available: http://www.microsoft.com/technet/prodtechnol/winxppro/proddocs/code_signing.asp (2002)
10. Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J., Deller, L., Reuther, L.: The SawMill multiserver approach. In: 9th SIGOPS European Workshop, Kolding, Denmark (2000)
11. Härtig, H., Baumgartl, R., Borriss, M., Hamann, C.J., Hohmuth, M., Mehnert, F., Reuther, L., Schönberg, S., Wolter, J.: DROPS - OS support for distributed multimedia applications. In: Proc. 8th SIGOPS European Workshop, Sintra, Portugal (1998)
12. von Eicken, T., Basu, A., Buch, V., Vogels, W.: U-net: a user-level network interface for parallel and distributed computing. In: Proc. 15th Symp. on Operating Systems Principles, Copper Mountain, Colorado, USA (1995) 40–53
13. Myrinet: Myrinet. Website: www.myrinet.com (2002)
14. Leslie, B., Heiser, G.: Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School Computer Science and Engineering, University of New South Wales, Sydney, 2052, Australia (2003)
15. Felten, E.W., Alpert, R.D., Bilas, A., Blumrich, M.A., Clark, D.W., Damianakis, S.N., Dubnicki, C., Iftode, L., Li, K.: Early experience with message-passing on the SHRIMP multicomputer. In: Proc. 23rd Symp. on Computer Architecture. (1996) 296–307
16. Rawson III, F.L.: An architecture for device drivers executing as user-level tasks. In: USENIX MACH III Symposium. (1993)
17. Liedtke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N., Jaeger, T.: Achieved IPC performance. In: 6th Workshop on Hot Topics in Operating Systems (HotOS), Chatham, Massachusetts (1997)
18. Aron, M., Liedtke, J., Park, Y., Deller, L., Elphinstone, K., Jaeger, T.: The SawMill framework for virtual memory diversity. In: Australasian Computer Systems Architecture Conference, Gold Coast, Australia, IEEE Computer Society Press (2001)
19. Liedtke, J., Uhlig, V., Elphinstone, K., Jaeger, T., Park, Y.: How to schedule unlimited memory pinning of untrusted processes or provisional ideas about service-neutrality. In: 7th Workshop on Hot Topics in Operating Systems, Rio Rico, Arizona (1999)
20. Unrau, R., Krieger, O.: Efficient sleep/wake-up protocols for user-level IPC. In: International Conference on Parallel Processing. (1998)
21. Ritchie, D., Neufeld, G.: User level ipc and device management in the raven kernel. In: Proc. USENIX Microkernels and Other Kernel Architectures. (1993)
22. Wienand, I., Macpherson, L.: ipbench. Website: http://ipbench.sourceforge.net/ (2002)