Factorization of Device Driver Code between Kernel and User Spaces

Arini Balakrishnan and Swetha Krishnan
Computer Sciences Department
University of Wisconsin, Madison
arinib,swetha@cs.wisc.edu

Abstract

Device drivers, which are normally implemented as kernel code, pose stability problems since bugs in the drivers cause kernel crashes. Running device drivers as unprivileged user-level code has often been proposed as a solution to increase the robustness of the system. However, moving the entire driver to user space brings down the performance of the system. An alternative approach would be to retain the performance critical code in the kernel and move the less performance sensitive code to the user space. In this project, we propose a scheme for factorization of driver code based on performance. In our split driver, work that needs to be done fast such as device I/O and interrupt handling is retained in the kernel space. Work that is less common and can afford to be done slower such as configuration or statictics collection is moved to the user space. We implemented this scheme on PCnet32 network driver and measured the performance overhead incurred by moving some of the driver functions to user space. We found the performance overhead to be less than a factor of 2. Also, the performance of the critical operations retained in the kernel was not affected by this factorization.

1 Introduction

Traditionally, linux device drivers are implemented as a part of the kernel source. They run in the kernel address space with kernel privileges and with full access to all system resources. This simplifies implementation and minimizes overhead. However, with the increasing number of devices, keeping all their drivers in kernel is leading to a rapid growth of the kernel code. More than 50% of the kernel code is made up of device drivers [9].

Managing in-kernel device drivers is also difficult. The in-kernel drivers have to adhere to the interfaces and conventions used by the kernel code. Drivers that run in the kernel have to updated regularly to match the in-kernel interface changes. As a result, drivers for uncommon devices tend to lag behind. Studies [3] show that in the 2.6.0-test10 kernel, there are 80 drivers known to be broken because they have not been updated to match the current APIs, and a number more than that are still using APIs that have been deprecated.

A recent study by Chou et.al [2] has shown that the defect density of device drivers is three to seven times that of other parts of the kernel. This is due to the inherent complexity of driver code and the fact that much of it is written by people not necessarily very familiar with the internal operating system structure. A crashing in-kernel driver often takes the rest of the system down with it. Also when drivers are implemented in the kernel, the driver faults can cause malfunctions in other unrelated kernel components. This makes fault isolation extremely difficult.

The microkernel approach to building kernels [6] proposes building drivers outside the kernel as unprivileged code. Ease of development is one argument for supporting user space device drivers. If a driver is a normal user process it can be debugged and profiled like any other user program. Fault source identification is also greatly simplified by removing the faulty drivers from the kernel and moving it to the user space. It also improves the system stability. When a user space driver crashes, the system is not rendered unstable since it can be killed and restarted like any other user process. But having entire device drivers in user space leads to a significant performance degradation even for the operations that are time critical.

Factorization brings the best of both these worlds by implementing a part of the device driver in the kernel and a part of it in the user space. We factorize the code based on performance, retaining in the kernel the performance sensitive code that is executed often. We move that part of the driver that is not performance sensitive to the user space. Thus our framework aims at making the common case fast.

The rest of the paper is structured as follows. Section 2 justifies the factorizing of device drivers by examining its advantages. Section 3 discusses the related work and section 4 discusses the design of the system. We discuss the implementation details of our split driver in section 5. We present our evaluation and results in section 6. We also discuss some ways to identify candidate functions i.e. those that can be moved to user space in section 7 and conclude in section 8.

2 Motivation

Factorizing the device driver code provides the advantages of the user-space device drivers and the performance benefits of the in-kernel device drivers.

Device drivers are written by people who are not kernel experts and so are prone to bugs. Though the drivers are written as kernel modules, the in-kernel driver code has access to the entire kernel address space, and runs with privileges that provides access to all the instructions. So bugs in device drivers can easily cause kernel panics and lockups. The only way to recover from such kernel panics is to reboot the system. Studies [2] show that 85% of bugs in the operating system are driver bugs. Factorizing device drivers helps in isolating the driver bugs from the kernel. It however does not isolate the bugs resulting from the part of the driver code retained in the kernel. The performance sensitive common path of the driver code that we decide to retain in the kernel is often well tested and is less buggy when compared to the less commonly executed code that we move to the user space. So, factorizing the driver would mean that there are less chances of a bug in the driver leading to a system crash.

Factorizing device drivers makes debugging easier. The portion of the device driver maintained in the user space can be easily debugged and profiled like any other user program. Also the user space portion of the device driver can make use of C libraries. For instance, we can use STL libraries to implement a queue or list in the driver. This is not possible in an in-kernel device driver.

Kernel memory is non-swappable, unlike user memory. So having drivers as in-kernel modules results in using up a lot of memory even for the infrequently used drivers. Factorization alleviates this problem by moving parts of the driver to the user space. Consequently, the infrequently used part of the device driver does not occupy memory when not in use.

With an in-kernel device driver, the kernel is kept busy even doing non critical operations of the device, such as getting statistics. This can be avoided with a factorized driver. Thus, the kernel can spend more time in doing useful work. While user space device drivers also relieve the kernel of these tasks, they additionally slow down even the critical driver operations.

One other benefit of factorizing driver code is the reduction in the amount of kernel code. This results in a very small kernel foot print. Factorizing thus provides a way to reduce the amount of in-kernel code without compromising on performance.

3 Related Work

The last two decades have seen substantial amount of research in addressing the problems posed by in-kernel device drivers. Most of the solutions involve moving the entire driver to the user space. Other solutions like Nooks [14] propose executing device drivers in light-weight protection domains within the kernel address space, that are isolated from the kernel. Our approach differs significantly from both these approaches. Firstly, we do not move all the driver code to the user space. Secondly, we also do not keep the complete driver code in the kernel address space.

Peter Chubb et. al [3] exploit the existing support in the Linux kernel and build on to it. The basic approach is to introduce new system calls that in turn invoke kernel routines. They implement a new file system using the PCI namespace and introduce a new system call that returns a file descriptor identifying a PCI device. This enables acquiring and releasing of a device by a single driver from user space. To enable DMA translations, a new system call is added that invokes the kernel functions to translate virtual addresses of buffers specified by user-level applications to I/O bus addresses. This call also invokes the in-kernel bookkeeping routines that ensure pinning of I/O buffers in memory. To deliver interrupts from devices to user-level drivers, interrupts are mapped to file descriptors and a new file(interrupt handler) is added to each possible interrupts directory in /proc/irq. When an interrupt occurs, the in-kernel handler disables the interrupt and increments a semaphore to wake up a user-level interrupt thread sleeping on the read() of the appropriate handler file. Finally, to pass work descriptors to the user-level driver from the kernel, and to offload work from the driver to the kernel, a shared memory region between the user and kernel address spaces is used, with lock-free queues to handle concurrency issues.

FUSD (Linux Framework for User space Devices) is a framework for proxying device file call backs into user-space and allows implementing device drivers as daemons instead of kernel code. It is a combination of kernel module and a user space library. The kernel module implements a character device which is used to control the channel between the two. The FUSD user space library provides functions for device registration. FUSD again attempts to move the entire device driver to the user space.

The user space approaches causes a slowdown of performance-critical events like acquiring and releasing of devices, handling interrupts, reads/writes and DMA translations. This is due to the overhead introduced by system calls and additional memory copies. On the contrary, our approach lets the kernel handle these operations to keep them fast.

Swift et. al [14] in Nooks achieve two goals namely fault-isolation and recovery from malfunctioning of device drivers. The approach taken here is to have the driver reside in kernel space, but within a separate protection domain. Nooks combines various approaches for fault isolation such as kernel wrapping of system calls, virtual memory protection and lowering of privilege levels and software fault isolation. By having the driver being kernel resident, they avoid overheads (such as copying of data between address spaces and translation of addresses) that affect performance in user-level drivers. The provision of various approaches provides adaptability to different environments and different device types. In our approach to factorizing device driver code, the part of the device driver in the kernel still runs in the kernel's memory address space. The user space part of the driver is completely isolated from the kernel's address space. We provide isolation based on performance. The part of the driver that is performance sensitive is often well tested and not prone to bugs. So, we find that we can achieve the isolation benefits by just moving the less commonly executed code to the user space.

4 Design

This section discusses the design of the system. The device driver is factorized into a $daemon$ that runs in the user space and a $kernel\ driver$ that runs in the kernel space. The system uses a simple $request-reply$ model for communicating between the kernel and the user space daemon. Incoming requests in the kernel for functions that have been moved to the user space are batched up and delegated to the user space daemon. The $daemon$ registers itself with the kernel to accept requests for the driver functions that are implemented in user space. After this registration, the user daemon thread waits in the kernel as a kernel thread. This is similar in semantics to LRPC [1]. When the waiting thread wakes up on being given a request by the kernel, the control returns to the user space. The $daemon$ processes the request i.e.

executes the requested function, returns the reply to the kernel and then processes the next request in the batch. If there are no more requests, it continues to wait in the kernel for future requests.

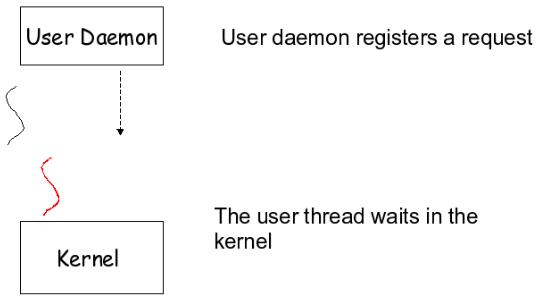

User Daemon
User daemon registers a request
The user thread waits in the kernel
Kernel

Figure 1: **Design - User Daemon Request.** *The user daemon registers with the kernel driver.*

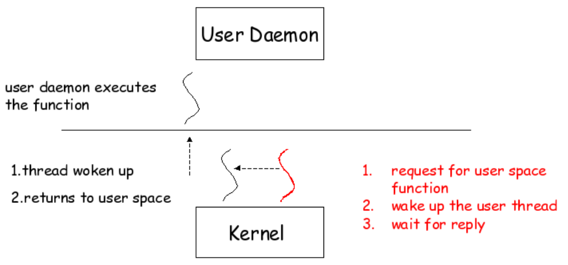

User Daemon
user daemon executes the function
1.thread woken up
2.returns to user space
1. request for user space function
2. wake up the user thread
3. wait for reply
Kernel

Figure 2: **Design - Kernel Reply.** *Kernel replies when the request arrives for a user space function*

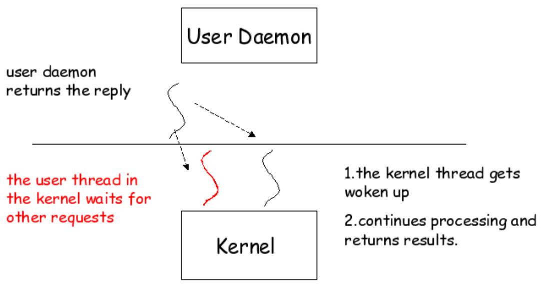

User Daemon
user daemon returns the reply
the user thread in the kernel waits for other requests
1.the kernel thread gets woken up
2.continues processing and returns results.
Kernel

Figure 3: **Design - User Reply.** *The user daemon replies with the result to the kernel and waits in the kernel*

The $kernel\ driver$ contains all the functions that are implemented in the kernel. Typically it has the functions that control I/O, device open, close and probe, the interrupt handler and the initialization and cleanup modules. The $kernel\ driver$ also contains a function that accepts a registration request from the $daemon$. When a request for a user space function is triggered, the $kerneldriver$ takes care of enqueueing the request and notifies the daemon of the request. The reply from the $kernel\ driver$ to the $daemon$ consists of the name of the function requested and the necessary arguments to execute the function.

Figures 1, 2, 3 gives the details of the design. They also provides a clear over view of what happens in each stage of the design.

5 Implementation

We implemented the factorization scheme discussed on the AMD PCnet32 network driver. This is the driver that the VMware Virtual Machine [8] supports, and we chose it so that we could move our implementation to VMware. On VMware, we could make kernel code modifications remotely, with an easy restart and recovery of the VM to a clean state following kernel panics. We used an emulation of the driver on a guest operating system running Linux 2.6.12.6 kernel on an Intel i386 architecture within VMware Workstation 5.5.1. The virtual machine had a 256 MB RAM , an 8 GB hard disk and one NAT-configured ethernet card. We chose Linux due to its support for drivers as loadable kernel modules, and due to its considerable existing support for user-mode drivers.

5.1 Existing Support

For moving functions to user space, we leveraged existing support in Linux in the following ways:

1. **System calls :** We needed the user daemon to call into the kernel to indicate it was ready to handle requests and also to pass back replies to the kernel after executing a requested function. This communication was achieved using the *ioctl()* system call. The *ioctl()* function performs a variety of device-specific control functions on device special files such as sockets. We defined our own control functions(commands) to be handled by pcnet32's *ioctl()* and performed ioctls from the user-space daemon with those commands. There were three such commands

namely COPY, to copy commonly needed kernel data structures, REQUEST, to wait in the kernel for a request, and REPLY, to pass back replies after handling a requested function.

To move functions doing reads and writes to the low-level I/O ports, we used the *ioperm()* and *iopl()* system calls. *ioperm()* gives a user process access to the first 1024 I/O ports, while *iopl()* changes the I/O privilege level of the process and can enable access(with level set to 3) from user space to all 65536 I/O ports. We used these calls in the user daemon along with *inb()*, *outb()*, *inw()* and *outw()* as required, to implement the I/O port read/write functions in the user daemon.

2. **Kernel API :** We needed to pass data back and forth between the kernel and user spaces, for sending function arguments to the user-space functions and for getting the results back into the kernel. This was achieved using the *copy_to_user()* and *copy_from_user()* functions provided in the Linux kernel API.

   We used semaphores to synchronize between the user thread making the ioctls into the kernel space and the kernel thread getting invoked upon incoming requests for a function either from another in-kernel function or from an external process. The linux semaphore API with *sema_init()*, *up()* and *down_interruptible()* functions were used for this purpose.

   Moving a function that uses a spinlock for mutually exclusive access to the device required further modifications. Such functions use the *spinlock_irq_save()* and *spinlock_irq_restore()* for disabling and re-enabling interrupts when the spinlock is acquired and relinquished respectively. This is done to prevent interrupt handlers from spinning forever if a device issues an interrupt while a function in the driver is executing with the spinlock held. However, if we simply moved what this function does to user space, leaving the spinlock in the kernel, it would result in the aforesaid race condition between the interrupt handler and the driver function since interrupts get re-enabled on switching to user space. To handle this, we replaced *spinlock_irq_save()* and *spinlock_irq_restore()* with the *disable_irq()* and *enable_irq()* functions of the API, and achieved the mutual exclusion using a mutex instead of the spinlock. *disable_irq()* and *enable_irq()* disable and enable the interrupt line of only our specific device, while keeping other interrupts enabled, so we can safely move the function's code to user space after grabbing the mutex, since in this case the specified interrupt line would not get re-enabled in user space.

5.2 User-Space Interface

We needed certain kernel data structures and constants of the driver to be visible in user space since they were accessed by the driver functions moved to user space. For this, we created a pcnet32.h header containing those structures in */usr/include/linux*. This file also contained a generic 'user_fn_args' structure that contained a function identifer and arguments of various data types that were required by various functions. The appropriate fields of this structure were filled in before passing it to the user space function through a *copy_to_user()* call.

There were also certain commonly needed helper functions(such as *pcnet32_wio_read_bcr()*, *pcnet32_wio_write_csr()*) doing reads and writes to I/O ports at specific addresses. These functions were being called both by the driver functions retained in kernel space and by the functions moved to user space. So we replicated these helper functions in our user-space daemon, so that they could be used by the user-space functions.

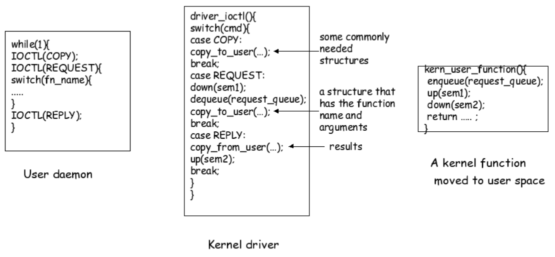
5.3 Handling Mutiple Requests

Consider the case when a request for a certain driver function that has been moved to user space is being processed. At this time, there would be no user thread waiting in the kernel since it is busy executing the function in user space. Now if there is a request for the same or another driver function that is in user space, then there would be no waiting thread in the kernel to receive it and so this request would be lost. To handle mutiple incoming requests while another is being processed, we created a FIFO request queue and enqueued every incoming request in it. Whenever the waiting thread was woken up, it would dequeue the request at the head of the queue and then switch back to user space to process it. This way, every request gets saved in the queue instead of being lost.

5.4 Sequence of Events

Figure 4 shows the code skeleton that is responsible for the control flow between the factorized parts of the driver. The chain of events can be summarized as follows:

1. First, the user space daemon issues an *ioctl()* with COPY as the command. This results in copying commonly needed structures, such as the device's private data structure, to user space.
2. Next, the daemon issues an *ioctl()* with the REQUEST command. This causes the user thread to wait on the semaphore in the kernel till any request comes in.
3. When a request is received, the kernel function meant to handle it just enqueues the request and then wakes up the waiting thread. It then itself goes to sleep on the second semaphore , waiting for the reply from the corresponding user space function.
4. The user daemon thread that was woken up dequeues the first request at the head of the queue and after copying the user_fn_args structure with the needed arguments to user space, it causes the REQUEST *ioctl()* to return to user space.
5. The user daemon invokes the function corresponding to the passed function identifier. After executing this function, it issues an *ioctl()* with REPLY as the command, passing the return value of the function as *ioctl()* arguments.
6. The kernel handles the REPLY *ioctl()* by copying the results to a kernel buffer and then waking up the thread that was sleeping on the second semaphore inside the kernel function. This thread then returns the results to the original caller.



```
while(1){
IOCTL(COPY);
IOCTL(REQUEST){
switch(fn_name){
.....
}
IOCTL(REPLY);
}
```

```
driver_ioctl(){
switch(cmd){
case COPY:
copy_to_user(...);
break;
case REQUEST:
down(sem1);
dequeue(request_queue);
copy_to_user(...);
break;
case REPLY:
copy_from_user(...);
up(sem2);
break;
}
}
```

```
kern_user_function(){
 enqueue(request_queue);
 up(sem1);
 down(sem2);
 return ..... ;
}
```

User daemon
Kernel driver
some commonly needed structures
a structure that has the function name and arguments
results
A kernel function moved to user space

Figure 4: **Code Skeleton.** *This is the main body of the framework that shows how the factorization is achieved.*

6 Evaluation

The PCnet32 driver has 54 functions (including the init and cleanup modules), out of which we moved 8 functions to user space. 16 helper functions were replicated in user space for being used by other functions. Section 6.1 outlines the rationale behind the functions moved. Section 6.2 details our test for correctness and section 6.3 discusses the performance of the driver both with and without moving these functions.

6.1 Functions moved to user space

We analyzed the driver code call patterns using cflow[1] to find out which functions were being referenced extensively and which weren't. This led us to identify 8 functions, which can be categorized as follows:

1. **'get' and 'set' methods:** *pcnet32_get_drvinfo()*, *pcnet32_get_ringparam* , *pcnet32_get_msglevel* , *pcnet32_set_msglevel* and *pcnet32_self_test_count*. These were methods that were being called by the *ethtool* [2] interface to acquire or change device settings such as rx/tx ring parameters, message level etc. We do not expect the ethtool commands to be run too frequently as compared to reception/transmission to and from the network device. So we felt these functions were good

[1]GNU cflow analyzes a collection of C source files and prints a graph, charting control flow within the program.

[2]Ethtool is a Linux net driver diagnostic and tuning tool for the Linux 2.4.x (or later) series of kernels. It obtains,for an ethernet device, information and diagnostics related to media, link status, driver version, PCI (or other) bus location, and more.

candidates to be moved to user space. These ethtool functions are commonly implemented by all drivers, so moving them to user space could be generalized to other drivers as well.

2. **Functions using spinlocks:** *pcnet32_get_regs()*. This is also an ethtool function to retrieve a register dump for the specified ethernet device. This function uses *read_csr()* mutiple times to read the contents of the control and status registers, the bus configuration registers and the mii physical registers. All this time-consuming work could be moved to user space since reading the I/O ports through *read_csr()* was duplicated in the user space too. Being a non-time-critical ethtool operation, this function too was suitable to be moved to user space, with the spinlock replaced by a mutex and *disable_irq()/enable_irq()*.

3. **Functions doing reads and writes to various I/O port addresses:** *mdio_read()* and *mdio_write()*. These functions read and/or write a specific number of bytes to I/O ports at specific offsets using *read_bcr()* and *write_bcr()*. Again, these functions were being used only by the mii-tool [3] and would not be invoked frequently.

As can be seen from Table 1, the total number of lines of code consisting of these functions moved from kernel to user space is about 100. This is not much, considering the size of the PCnet32 driver(2000 lines). However, there were certain functions that we could have moved but decided not to since mii support was not enabled in PCnet32 and these functions performed useful work only on mii being enabled. Also, most of the other functions were critical ones, doing device open/close, receive/transmit, handling interrupts and so on.

6.2 Correctness

To test our framework for correctness, we executed the ethtool commands. We executed these commands with our scheme i.e. with the user-daemon running and compared the results with those obtained without factorization i.e. with the ethtool functions retained in the kernel. The same results were obtained in both cases. Thus, running a piece of code in kernel and the other piece in user space does not affect the driver functionality.

6.3 Performance overhead

We measured the performance of the functions retained in the kernel and those moved to the user space. For the latter functions, we also measured the amount of time spent exclusively in user space operations. These measurements were done on the VM within VMware and hence would also include common virtual machine overheads. Deploying the factorized driver on actual hardware would give more precise measurements.

Table 2 gives the time taken to execute some of the ethtool commands with the in-kernel driver and with our factorized driver. As expected, there is a significant increase in time when the functions are executed in user space. When executing in user space, a process has normal privileges and can't do certain things. When executing in kernel space, a process has every privilege, and can do anything. Consequently, anything executed in user space would need to interact with the kernel and hence would be slower than if executed fully in the kernel domain. However, since the functions we moved are not the frequently used ones, the overhead does not affect the common-case performance of the driver. Moreover, the figures show that the increase in time is less than a factor of two.

The $-i$, $-g$ and $-s$ options shows a very small difference of around 0.002 seconds between the original driver and the factorized driver. On the other hand, the time taken for the $-d$ option differs by 0.08 seconds. This is because the former options get/set limited information and hence do not spend much time in copying between kernel and user space. However, $-d$ requires a regsiter dump of the contents of approximately 335 registers which involves port read/writes and also need more data to be copied back to the kernel. In case of $-t$, an entire ethtool test of the device is involved (which involves calling other functions like *pcnet32_ethtool_test()* and *pc-*

[3] mii-tool is a utility that checks or sets the status of a network interface's Media Independent Interface (MII) unit. It is used for determining if you are connected to the Ethernet, and if so, at what link speed and duplex status.

Source Components	#Lines of Code
Driver functions moved to user space	102
Driver functions replicated in user space	74
User space interface (header file)	254
User daemon	788
Total number of lines of code	1218

Table 1: **Number of non-comment lines of code added for factorization**

Command	Function	Original driver(in s)	Factorized driver(in s)
ethtool -i	get_drvinfo()	0.006	0.008
ethtool -g	get_ringparam()	0.007	0.009
ethtool -s	set_msglevel()	0.005	0.007
ethtool -t	self_test_count()	0.208	0.214
ethtool -d	get_regs()	0.245	0.332

Table 2: **Time taken for different ethtool operations**

Command	Original driver(in s)	Factorized driver(in s)
ping	9.568	9.245
scp	2.238	2.277
ssh	3.736	3.486

Table 3: **Time taken for network operations**

Command	COPY(in s)	REPLY(in s)	PROCESSING(in s)	Total time spent in user space(in s)
ethtool -i	0.000034	0.000085	0.000003	0.000122
ethtool -g	0.000025	0.000137	0.000002	0.000162
ethtool -d	0.000056	0.003862	0.271508	0.275426
ethtool -s	0.000025	0.000109	0.000081	0.000215

Table 4: **Time split up in user space**

net32_loopback_test()) and only a small portion of the operation, *pcnet32_self_test_count()*, which simply gived the test length, has been moved to user space. Hence the time difference noticed is only 0.006 secs.

Table 3 gives the time time taken to execute some network commands like $ping$, scp, and ssh, that call the functions retained in the kernel. These functions would involve performance-critical operations such as reception/transmission of packets. We executed scp by copying a 4.3 KB file from the host machine to our VM. We timed ssh by logging in to the host from the VM and exiting immediately. The figures show that there is hardly any difference in time in these operations when executed with the original and factorized drivers. This is expected because we did not move any of the time-critical functions to user space. Thus, factorization does not slow down the critical operations retained in the kernel.

Table 4 gives the time spent in the different phases namely COPY *ioctl()*, REPLY *ioctl()* and request processing, in user space. The $-d$ option involves reads and writes to the I/O ports to get the register dump. This is the reason for the significant time (0.275426 secs) being spent in user space. On the other hand, the other options do not have much to do and hence spend very little time in user space.

7 Identifying Candidate Functions

To come up with a standard for identifying the functions that can be moved to the user space, we analysed two other network drivers $e100 - IntelPro/100ethernetdriver$ and $e1000 - IntelPro/1000Gigabitethernetdriver$ using cflow. We also plotted the call graph output from cflow with dot[4]. The clustered regions in the dot graph helped us identify the functions that were isolated from the performance sensitive code (like *pcnet32_rx()*, *pcnet32_start_xmit()* etc.). Based on this analysis we came up with the following general rules for identifying functions in a network driver that must be moved to the user space and those that can/might be moved to the user space.

[4]dot is a tool that draws directed graphs as hierarchies.

'Must' candidates: All the ethtool operations are ideal candidates, since they are called only by ethtool. PCnet32 supports only few such operations, while e100 and e1000 implement a large number of them and allow more scope for movement.

The *mdio_read()* and *mdio_write()* operations in these drivers can also be moved, since they are called only by the mii interface. These functions do reads and writes to I/O ports and moving them would relieve the kernel of the time spent in these non-critical functions.

There are also a number of boolean methods that perform validations or check certain conditions. e1000 has some such methods such as *e1000_check_64k_bound()* and *e1000_validate_options()*. Again, these methods are not performance critical and hence can be moved to the user space.

Along the same lines, functions such as *e100_set_multicast()* and *e100_change_mtu()* can be moved since the muticast list or mtu would not be changed too often .

'Might' candidates: NIC configuration functions like *e100_configure()*, *e1000_set_phy_type()* are invoked only during device configuration which would normally be done rarely. Depending upon whether or not they require access to the hardware directly, we might or might not be able to move them to user space.

'get' and 'set methods such as *e100_get_defaults()*, *e100_get_stats()*, *e100_update_stats()* and *e1000_get_phyinfo()* can be moved depending upon how frequently they are being invoked by external processes. For instance, if many critical network operations also do a read of the network device statistics, it would not make sense to move a *get_stats()* function to the user space since that would increase the overall time for the critical function.

8 Conclusions

Our implementation and evaluation of the proposed factorization scheme on the PCnet32 driver shows a proof of the concept that infrequently used driver code can be moved to the user space without a massive performance overhead. This means the kernel is isolated from any driver crashes that might be caused due to the functions that are now in user space. The performance of the sensitive code retained in the kernel does not suffer any degradation with the factorized driver.

The gain obtained by this approach depends to some extent on how much scope the driver provides for 'must-move' candidate functions and upon the extent of work that is being delegated to user space. The e100 and e1000 drivers are rich in such candidate functions as compared to PCnet32. Moving functions like *get_regs()* relieves the kernel of a lot of time-consuming tasks.

What we have shown is a manual approach to factorization by investigating network driver code. Our work can be extended to develop a tool to automatically factorize drivers based on the general rules identified. Also, it would be interesting to investigate other drivers such as char drivers or block drivers to see how factorization can be applied to them.

References

[1] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 102–113, New York, NY, USA, 1989. ACM Press.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[3] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symposium*, July 2004.

[4] P. Chubb. Linux kernel infrastructure for user-level device drivers. In *Linux Conference,Adelaide,Australia*, January 2004.

[5] J. Elson. FUSD: A linux framework for user-space devices, August 2003.

[6] A. Forin, D. Golub, and B. Bershad. An I/O system for mach 3.0. In *Usenix Mach Symposium, 1991*, pages 163–176, November 1991.

[7] G. C. Hunt. Creating user-mode device drivers with a proxy. In *First USENIX Windows NT WS,1997*, pages 55–59, 1997.

[8] J.Sugerman, G.Venkitachalam, and B.Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, June 2001.

[9] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gtz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. Technical report, PA005043, National ICT Australia, July 2005.

[10] B. Leslie, N. FitzRoy-Dale, and G. Heiser. Encapsulated user-level device drivers in the mungi operating system. In *Workshop on Object Systems and Software Architectures*, January 2004.

[11] B. Leslie and G. Heiser. Towards untrusted device drivers. Technical report, UNSW-CSE-TR-0303,School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, March 2003.

[12] K. V. Maren. The fluke device driver framework, 1999.

[13] A. Rubini, J. Corbet, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly.

[14] M. Swift, S. Martin, H. M. Leyland, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In *Tenth ACM SIGOPS European Workshop,Saint Emillion, France, September 2002*, September 2002.

[15] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.