Simultaneity Safari:
A Study of Concurrency Bugs in Device Drivers

Markus Peloquin, et al.

Department of Computer Science
University of Wisconsin–Madison
`markus@cs.wisc.edu`

December 21, 2009

Abstract

Concurrency has been an issue in device drivers since the advent of interrupts. The ubiquity of multi-core CPUs increases the urgency to fix errors in the code (since the probability of manifestation increases) and to understand the issues involved as architectural priorities change (decreasing the probability that threads block). The particular challenges associated with concurrent programs, however, have not been handled appropriately in many cases. Since the development of tools to accurately test and debug programs that use concurrency hinges on a deeper level of understanding of concurrency bugs than currently exist, this paper seeks to advance the state of the art by outlining real world examples of these bugs as they appear in device drivers.

We have carefully examined the origins, scopes, and mitigation strategies for a collection of 98 real world concurrency bugs drawn from a pool of USB, FireWire, and PCI device drivers. Our study reveals several significant classes of bugs and a selection of interesting anomalies, both of which may be useful in avoiding or addressing concurrency bugs in other situations. Specifically, we found that: (1) Around 20% of the bugs we examined were deadlock bugs, the majority of which could be grouped into certain faulty assumptions which, once recognized, could be avoided in future development; (2) Of the non-deadlock concurrency bugs, nearly 43% were caused by disconnection event or power management handling sections of the drivers. This indicates that the hot-swappable functionality provided by the USB and FireWire protocols entails a certain challenge particular to device drivers; (3) Although ensuring proper locking protocol (esp. lock balance and lock initialization) is a well-studied issue in concurrency bug testing, a full 10% of all bugs examined in this study failed to correctly manage their locks.

Keywords: driver concurrency, concurrency bug, bug characteristics, USB, FireWire, PCI

1   Introduction

The myriad of bugs present in device drivers has led some to claim that device drivers, as a group, are the single biggest threat to kernel reliability [1, 3, 7]. Threats imposed by failing drivers include data loss, memory corruption (with subsequent kernel panics or instability), and storage corruption. Some techniques to make drivers safer involve hardware memory protection, privilege separation, and the usage of safe languages [11]. Previous approaches have been proposed that detect and test user-level concurrent programs [5, 9, 12].

For this project, we examined a number of concurrency-related bug fixes in Linux device drivers. By examining the nature of these bugs, we hoped to

gain a deeper understanding of how device drivers differ differ from applications, how concurrency bugs are introduced, and how concurrency bugs can negatively impact kernel reliability. We hoped that this would give insight into approaches for detecting and avoiding bugs in device drivers in the future.

1.1 Motivation

Driver code makes up 70% of the Linux codebase, making drivers an important target for preventing and fixing bugs. In addition, drivers are more buggy than other parts of the kernel; Chou et al. find that the vast majority of bugs in the Linux kernel occur within driver code, and that the rate of bugs is several times higher than for other parts of the kernel [1]. Driver bugs can also have drastic effects; bugs in driver code account for a significant percentage of the crashes in a system. In Windows XP, driver bugs cause 85% of system crashes [10].

Many of the driver bugs are related to concurrency. A recent work studied the development history of representative drivers and found that concurrency bugs account for 19% of device driver bugs [8]. While other studies have explored concurrency bugs found in server and client applications [6], this paper provides the first comprehensive look at concurrency bugs found in device drivers. It is the intention of the authors that this study can be used to reveal interesting findings and provide useful guidance for concurrency bug detection and resolution.

It is important to understand concurrency bugs for several reasons. By understanding them, it may be possible to avoid introducing them. If it is noted that the majority of the bugs resemble one another, it may be possible to avoid them in the future by knowing to check similar code closely. It might also make it possible to fix existing, undetected bugs by looking for similar cases. For example, if a number of concurrency bugs involving the same function are found, it is an indication that in the future, developers should be especially careful writing code that includes that function. In addition, places where that function occur should be considered automatically suspect if a bug is encountered.

The intent of this project was three-fold: to ex-plore the device driver environments to determine what form concurrency bugs take in existing drivers; to determine how those bugs were introduced to their drivers; and to determine what, if any, general steps can be taken to prevent the unforeseen introduction of these errors in device drivers in the future.

2 Methodology

A list of bugs in several kinds of device drivers (USB, Firewire, PCI) in Linux was obtained from the authors of the Dingo paper [8]. The list contained references for a large sampling of bugs from a variety of device drivers. Each entry contained a pointer to a BitKeeper page with changeset details, a comment from the author, and the author's classification of the bug.

Using this list as a starting point, we examined each bug by hand to determine what similarities and differences existed between the bugs. The resulting documentation included a description of the bug, a description of the fix, and, when available, a history of the bug and the consequences of its manifestation. After processing the entire list of bugs, we designed classifications for the prevalent patterns and interesting anomalies that we had encountered. These classifications segregated the bug list according to such features as potential manifestation locations and what methods were used to detect and repair the bug.

2.1 Bug Sources

Linux's source code management was done with Bit-Keeper from 2002 to 2005. Though Linux switched to Git at version 2.6.12, BitKeeper currently contains changes up to release 2.6.28. As such, BitKeeper is the best source of "changeset" descriptions. Each changeset includes a description of what the patch was supposed to do, as well as a simple interface for displaying the changes and an annotated version of the source showing in which revision each line was added. The descriptions by the submitter of the patch were very helpful in determining the intent of the author.

2.2 Driver/Bug Selection

The USB/FireWire devices were interesting drivers to examine for concurrency bugs due to the high correlation between the drivers of outwardly dissimilar devices. Since both USB and FireWire peripherals support plug-and-play and hot-swapping paradigms, the drivers must be able to handle similar special functionality including graceful handling of unanticipated disconnect events; in fact, of the observed bugs, races during disconnect events were the single largest source of bugs in USB and FireWire drivers (see Figure 1). As similar robustness is necessary with power management handling, this further validates the selection of device drivers as interesting subjects for our concurrency bug hunt.

The PCI drivers we studied included drivers from network, frame buffer, sound card, and host channel adapter card devices. While these devices lacked some of the commonalities of USB/FireWire device drivers, they provided another avenue of exploration that included such interesting challenges as bug manifestations as a result of unanticipated CPU re-orderings.

As an additional avenue of exploration, we chose a selection of bugs from various sources as candidates for replication. These bugs were selected specifically as they seemed the easiest to reproduce and were documented as manifested bugs in their accompanying literature. Emphasis was also placed on bugs that the team felt were either exemplars of common bug situations or were very well-understood by the team. The motivation sustaining the replication effort was that of attaining a deeper understanding of some of the difficulties inherent in debugging manifested concurrency bugs.

3 Bug Classifications

For the bug patterns, we classified each bug into three primary categories: *locking protocol*, *deadlocks*, and *races*. These and other, more specific, categories are shown in Figure 1 for our drivers, as well as described in the following sections.
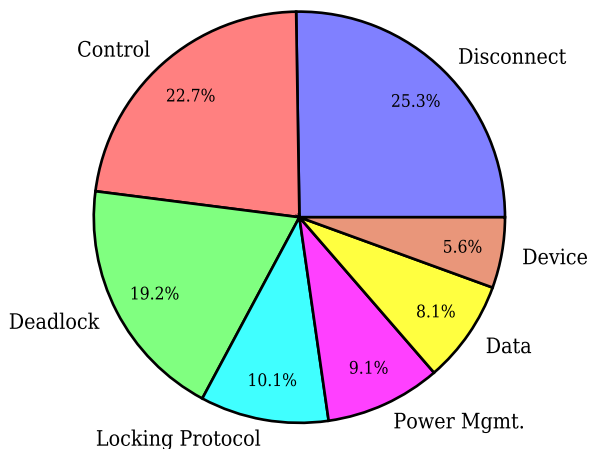

Figure 1: Types of Device Driver Bugs

3.1 Locking Protocol

For the purposes of this paper, a locking protocol bug is any bug that fails to properly utilize the locks it has. The two main subcategories of *locking protocol* bugs are *lock balance* and *spinlock API*.

3.1.1 Lock Balance

A *lock balance* bug is any bug where a lock is either held but never released or is released but never previously held. While these bugs could have easily been detected using some automated method (e.g. using a novel template with the methods described by Engler et al. [2]), no comment was made in any of the observed examples that indicated how the bug had been detected. Even with the relative simplicity of bugs in this category, we encountered three instances of this bug, two undetected for thirteen months [Mellanox-10,13][1]. This shows how important it is to employ

---

[1]This notation indicates an exact bug reference. For our notes on it, see the paper's wiki at http://cs736concurrency.wikia.com/.

automated analysis techniques as a supplement to reactive analysis.

3.1.2 Spinlock API

A *spinlock API* bug manifests when an attempt is made to use the lock without having previously properly initialized it. Again, automated analysis methods could probably have been used to detect these bugs; that they remained undetected for one year, on average, is evidence that automated analysis techniques were not used.

3.1.3 Other

Within the locking protocol category, there were some bugs that we determined were clearly related to a failure in the locking protocol, but which did not fit into any other category. Such bugs included an attempt to acquire the same lock multiple times in a single thread and an attempt to acquire a series of locks from different contexts and in different orders. These bugs had varying severity, some resulting in machine freezes. In this group, one deadlock (which was caused by improper lock acquisition ordering) was detected using Lockdep [Mellanox-06]. The nature of these locks strongly suggests that others within the group might be detectable in a similar manner.

3.2 Deadlocks

A *deadlock* is a bug where a race condition could cause the code to deadlock indefinitely. While all deadlocks are races, we have separated these from the race category as they are a special case. Accordingly, a bug that is reported as a deadlock is not further reported as a race in another category. In the course of this study, the three main subcategories that we encountered were *memory allocation*, *non-interrupt-safe behavior*, and *unsafe unlink*.

3.2.1 Memory Allocation

A *memory allocation* bug is a specific bug related to improper use of the `GFP_KERNEL` flag during a memory allocation attempt. The `GFP_KERNEL` flag was added to certain memory allocation functions common to many of the drivers we studied. The flag, as relevant to the purposes of the bugs we encountered, was used to allow the programmer to signal whether the memory allocation needed to happen in an atomic context (`GFP_ATOMIC`), in a non-atomic but I/O-free context (`GFP_NOIO`), or a regular context (`GFP_KERNEL`).

This flag was added as an argument to the function `usb_submit_urb()`, where it had previously defaulted to `GFP_KERNEL`. When the argument was added, all calls to it began using kernel allocation explicitly. This was a bit of a roadblock to fix errors in various USB drivers, since they could not control how memory was allocated.

Two major bugs were able to be fixed as a result: a function called in an atomic context could then block in the memory allocation step, and a function called when I/O operations are prohibited could be subjected to I/O operations as a means of satisfying the memory allocation request. Of all the bugs encountered in the course of this study, the memory allocation bugs comprised the largest group. They serve as an example where a single, broad-sweeping assumption can negatively impact the reliability of device drivers across the board.

3.2.2 Non-Interrupt-Safe Behavior

A *non-interrupt-safe behavior* bug is a more general bug related to the improper use of unsafe interrupt behaviors in a context where interrupts could occur. For the most part, these bugs manifest in similar situations as the memory allocation context-related bugs described above, but occur in a different context than the aforementioned memory allocation flags. One of these bugs, related to a non-interrupt-safe spinlock being used in an interruptible context, was identified by Lockdep.

3.2.3 Unsafe Unlink

An *unsafe unlink* bug is a specific non-interrupt-safe behavior bug that came up multiple times. To cancel a USB Request Block (URB), one might use the `usb_kill_urb()` function, but this is guaranteed to

block. The alternative is to use `usb_unlink_urb()`.

This bug manifested twice. Once was in older code, at a time when the kernel required a flag to be set in the URB ahead of time, so it was easy to forget [USB-Net-02]. When the flag was finally removed from the kernel, it was clear that a few developers had made the mistake of omitting it at some point. The other manifestation was caused by a developer changing the *unlink* version to *kill*, and reverted moments later by the driver's primary developer [RTL8150-05].

3.3 Other Races

A *race bug* is any race-related bug that does not cause a deadlock condition. As these bugs do not share a common characteristic in many cases, they are instead classified according to the context in which they occur. As such, the subcategories of race bugs are *control*, *data*, *device*, *disconnect*, and *power management*.

3.3.1 Control

A *control race* is any race that occurs in a control path of the device driver during the connection, initialization, or normal use of the device. Control races do not include races that occur during device disconnections or power suspensions. These errors varied significantly in nature but included some patterns. Some common behaviors included failure to protect a variable from non-atomic access patterns, prematurely registering or de-registering a device in a non-atomic context, and failure to protect paths where the driver could proceed down two mutually exclusive paths or down the same non-reentrant path from multiple contexts.

3.3.2 Data

A *data race* is a race that occurs in the data path of the device driver. These bugs were highly dissimilar from one another and shared no common features beyond the nature of their manifesting contexts.

Control-Data *Control-data races* are simply races between execution of control and data paths. We only encountered one such bug in the RTL8150 driver, where a queue remained active after the traffic had been disabled and the URBs had been asynchronously unlinked [RTL8150-03]. The bug was fixed in the control path alone. In this study, the lone control-data race is counted as both a control race and a data race for reporting purposes.

3.3.3 Device

A *device race* is a race that manifests during a device's direct memory access. Within this subcategory, two-thirds of the bugs observed occurred in the Mellanox InfiniHost III Infiniband host controller. These bugs were particularly interesting in that, for the most part, they manifested as a result of CPU re-ordering.

The CPU re-ordering bugs encountered in device races were highly dissimilar and largely dependant on the assumptions of the developers. As an example, one bug manifested when on systems with multiple CPUs issuing simultaneous commands to the device [Mellanox-14]. What could happen is that the commands, though strictly ordered, in theory, by a mutex in the code, could be re-ordered by the CPU and arrive out of order at the device. To fix this bug, and most other CPU re-ordering bugs encountered, the simple solution of adding a memory I/O write barrier was sufficient.

Device-Data *Device-data races* are another rare subclass of races, and are characterized by a race between the data path and the device. The only example we encountered involved a breakdown in coherency for a cached table, though other examples are imaginable [Mellanox-08]. In this study, the lone device-data race is counted as both a device race and a data race for reporting purposes.

3.3.4 Disconnect

We classified as *disconnect races* any control race that occurred during the device disconnection events. Such events could include bugs that manifest due to hardware-initiated, hot-swapping disconnect events

or due to driver-initiated, component-shutdown disconnect events. Many of these bugs had to do either with the attempted use of a recently de-allocated resource or the attempted processing of an invalid work-queue event. One such example is presented in Section 5.2.1.

3.3.5 Power Management

The final class of races, *power management races*, occur when the device is placed into or later removed from a low-power hibernation mode as a power management technique. These bugs were highly dissimilar from one another and shared no common features beyond the nature of their manifesting contexts.

3.4 Uncategorizable

Despite our best efforts, there were some bugs for which we could not determine a satisfactory classification. These bugs included two from the Kawasaki LSI KL5KUSB100 USB to Ethernet controller driver and one from the USB core hub driver [KL5K-05]. They were parts of significantly larger commits that made changes to a sufficiently large number of locations, so determining the root cause of the bug that was being addressed was nigh impossible. Were the changelogs more descriptive, the changesets smaller, or the code better commented, it would have been possible to learn something from these bugs. Unfortunately, since few coders see the inherent value of code readability, it is highly likely that there will always be bugs whose fixes escape all but the original programmer.

4 Bug Replication

The ability to reproduce bugs is beneficial for a couple of reasons. It confirms understanding of what the fix is actually doing. It also would serve as a test case for bug detection tools.

We used several approaches to reproduce driver concurrency bugs. Actual hardware was obtained, and multiple attempts were used to create the conditions "fixed" by some of the developers.

4.1 Reproducing a Crash Report

One of the devices that seemed to have a rather nasty error was the RTL8150. The changelog for indicates that it "crashes the kernel if the USB lead is unplugged while the device is active" [RTL8150-07]. Reproduction should only require producing a load on the device before unplugging it. Unfortunately, outdated sources led to our conclusion that the device we chose did not have the same, but a similar chipset. As such, it had no disconnect bug.

4.2 Reproducing Memory Allocation Errors

The other approach was to modify working code to reproduce the more popular memory allocation bugs. One attempt was made to cause I/O in memory allocations in the Device Mapper (a change from GFP_NOIO to GFP_KERNEL). The broken Device Mapper was fed a stream of I/O and control operations for four hours without failing.

The next point of attack was the Human Interface Device driver. Whenever the mouse moves, an interrupt is fired that will allocate memory with GFP_ATOMIC. This was changed to do kernel allocation requiring a far larger amount of memory, and the amount allocated would increase at each interrupt. No noticeable problems were encountered after causing a quarter million interrupts.

It appears that the memory allocation errors deserve the title *hypothetical*. It is a reasonable precaution, as these flags appear over three thousand times in the kernel, and they cannot all be unnecessary. Supposing they do actually occur, the window is probably too small in most cases.

5 Bug Extermination

The changes came about for a few reasons. Bugs may be detected by automated tools or reported by users. There are other cases where the change is instigated by a careful developer, or for no apparent reason. The causes of the bug fixes are shown in Figure 2 and in the following sections.
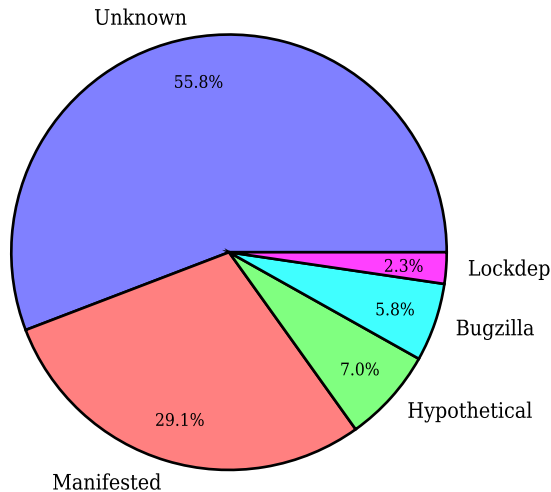
Figure 2: Detection Method of Device Driver Bugs

5.1 Bug Detection

We looked at the techniques used to find bugs in the first place. In most cases, the changesets did not document how the bug was discovered. However, there were some cases where the exact method was noted. Documentation included links to Bugzilla entries, comments that Lockdep was used, explicit acknowledgements that the bug was seen in the wild, notes that the bug was hypothetical, and references to "fixed" races or crashes. This last was problematic in that it was often unclear about whether the buggy behavior had ever actually been observed, or whether the fix was simply aimed at preventing it.

Several bugs in the Mellanox driver were detected using Lockdep. Lockdep is an optional validator in the Linux kernel. It detects behavior that can lead to lock inversion (a cycle in a partial ordering). There are also checks whether conflicting *safe* and *unsafe* types of locks are held at the same time. It proves with complete certainty whether a lock-based deadlock is possible regardless of what happens with the scheduling. It seems to leave the door open, however,

for deadlocks caused by blocks in atomic contexts.

Some of the changesets we examined included Bugzilla reports. These indicate that the behavior was observed by users who were not part of the development team, and that the effects of the bugs were sufficiently severe that these users filed bug reports. These mainly seemed to involve easily reproducible crashes.

A few changesets indicated that the bug had never been seen to manifest; for example, our notes for a bug in the USB network infrastructure indicate notes that while they do not know of any problems that were definitely caused by the bug, they still think that it was worth fixing before a release [USB-Net-06].

For the majority of changesets, there was not even a hint about how the bug might have been detected. It is likely that many of these actually did manifest, which is what drew the developers' attention to them; however, it is also possible that some of them were just detected by developers who happened to notice potential problems, or who were specifically looking for them (for example, because they knew that GPF_KERNEL was often used inappropriately).

5.2 Bug Fixes

Depending on the type of bug and the circumstances where it occurred, there were a variety of approaches to fixing driver concurrency bugs. In some cases, the fix was as simple as adding locks where they had not previously been, initializing a previously uninitialized spinlock, or switching from potentially blocking functions to guaranteed non-blocking functions. In some cases, the code was rearranged to avoid the issue entirely—for example, by moving the call of a potentially blocking function to a point after the lock was released as in the USB network infrastructure [USB-Net-04]. The exact way of fixing the bug proved highly dependent on both the type of bug as well as its effects.

5.2.1 Bug Non-Fixes

One can hardly say that a bug has been fixed if it has never occurred. Concurrency is far too unpre-

dictable, and the infinite interleavings possible (esp. on multi-core systems) demand a higher level of assurance. We commented on a classic example of bug non-fixing in Section 4.2. The only indication that any of these bugs actually occurred was one of the first to be fixed, where the changelog used language like "fixes . . . deadlock" [KL5K-08].

There are cases where the change is only to reduce the possibility of a race; one such case can be found in the USB serial converter driver [USB-Serial-10]. For this bug, the portion of the code where the change occurred was removed thirteen months later by the primary developer of the driver without any word of the race. In some cases, reducing the probability that a bug would manifest (window shrinking) may be considered a valid fix, especially if a truly "correct" fix would be very complicated or might hurt performance in the common case. However, we did not see many cases where window shrinking was chosen as the method to fix a bug.

5.2.2 Incorrect Fixes

There were a few cases of incorrect fixes. The time to fix such bugs ranged from one hour to six months. The one that escaped notice the longest was in the USB serial converter driver [USB-Serial-02]. In this bug, a spinlock was held while calling a blocking function. The fix was to replace the spinlock with a mutex; since spinlocks disable thread switching, they cannot be used across blocking code. We could not find documentation describing this in the Linux kernel.

5.3 Bug Avoidance

Sometimes code is changed not because it was broken, but because it could be written in a way that encourages safety. This happened once in the USB storage driver [USB-Storage-10]. The SCSI subsystem has three layers, which lends itself naturally to a locking convention. A semaphore was introduced that covered only the middle layer. Now the locks can be layered in the same way as the architecture, and deadlocks become much harder to create.

6 Related Work

Related works have previously explored the interplay between device drivers and concurrency issues. Dingo is an event-driven architecture for device driver development [8]. It focuses on protecting against two classes of bugs: concurrency faults and software protocol violations (failure to properly interact with the OS). Ryzhyk et al. made the observation that drivers lend themselves to an event-driven implementation more so than a multithreaded implementation. Synchronization becomes much more localized and easier to work out, and much of this is handled by Dingo, itself. Protocol violations are prevented with the use of *protocol specifications*. Drivers declare in their *component specification* which protocols they support. A single protocol implementation is then shared among all drivers that need it in a uniform way.

Some work has been done to make drivers more resilient to general failures. The Nooks architecture for Linux provides a wrapper for drivers, giving multiple levels of protection, while requiring trivial changes to existing drivers [11]. This includes lowering privilege levels and emulating a protection domain. Corruption of external data structures is prevented, but corruption of internal data and deadlocks still can occur. The Minix 3 OS isolates drivers within user-mode processes, and so shares most functional advantages of Nooks [4]. It further allows for problematic drivers to be detected and restarted as necessary. Heartbeats emitted by the driver are used as an indication of deadlocks, causing a driver restart.

Several techniques have been used to detect and even prevent races. Eraser is a run-time analysis tool that detects many common data races [9]. It is able to study the execution by modifying the binary. Dimmunix is another run-time analysis tool, and is able to both detect and prevent deadlocks from occurring [5]. It is implemented by adding hooks to the threading library, so it is probably easier to implement in the kernel than Eraser. A third tool is DDVerify, which is a static analysis tool that specializes in detecting synchronization errors [12].

Chou et al. looked at errors in the Linux kernel which were detected automatically by compiler extensions. They found that drivers accounted for be-

tween 70% and 90% of bugs, depending on the type of bug. They also looked at other characteristics of bugs, such as bug density by function length, bug distribution across files, and the lifetimes of bugs [1].

7   Discussion

There were several characteristics we noticed about Linux device driver concurrency bugs that might be useful for developers trying to detect or avoid similar bugs in the future. First, we noticed that there are several very common causes of problems. For example, the incorrect use of GFP_KERNEL was responsible for a large number of deadlocks, indicating that driver developers should take a careful look at any code using the flag to make sure it is appropriate. In general, calling potentially blocking functions from atomic contexts was a bug that came up repeatedly in our examination of changesets; appearing in such contexts as memory allocation, *unsafe unlink*, and work queue bugs.

A type of bug that surprised us with its frequency was locking protocol bugs. 10% of the changesets we examined dealt with fixing a bug caused by improper locking protocol. We encountered several that were very simple: (1) forgetting to initialize a spinlock or (2) entering the function epilogue responsible for unlocking locks taken by the function, even when the locks were never taken. These bugs indicate that even simple static analysis might be worthwhile for device drivers.

7.1   Limitations

There are several limitations to our results that are inherent to the way that the set of bugs we examined were chosen. Because all of these bugs were found by looking in the changesets for fixes, the sample of bugs we have might not be representative of driver concurrency bugs as a whole. It may be biased towards bugs that manifest relatively frequently, bugs with severe results, and bugs that are simple to fix, because these are characteristics of bugs that would make them more likely to be fixed. It is possible that there are numerous other concurrency bugs in the

drivers which we did not see because either the effects were not very noticeable or because they did not occur frequently enough to be reported and fixed.

7.2   Future Work

There are several things that were outside the scope of our project that would be interesting to investigate. As mentioned in Section 7.1, our results might be affected by a bias in the bugs we were able to examine. Although it would be hard to completely avoid this effect, looking to see the characteristics of bugs discovered by other methods, such as by static analysis, might yield further insights into the characteristics of device driver bugs.

It would also be interesting to see how Linux device drivers compare to bugs in drivers for other operating systems, although this would likely be difficult due to the closed-source nature of many Windows drivers. Another question to investigate might be the similarity of concurrency bugs found in device drivers to concurrency bugs found in other domains.

8   Conclusions

This paper provides a comprehensive study of a number of real world concurrency bugs in device drivers and provides a study of their characteristics, manifestations, detection methods, and fixing strategies. The study references 98 concurrency bugs drawn from a pool of USB, FireWire, and PCI drivers across multiple device domains. Included in the examination were a number of bug paradigms as well as a selection of unique, yet interesting, bugs in the hope that their study may drive future work in concurrency bug extermination efforts.

9   Acknowledgements

References

[1] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM, pp. 73–88.

[2] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001), 57–72.

[3] GANAPATHI, A., GANAPATHI, V., AND PATTERSON, D. Windows xp kernel crash analysis. In *LISA '06: Proceedings of the 20th conference on Large Installation System Administration* (Berkeley, CA, USA, 2006), USENIX Association, pp. 12–12.

[4] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Failure resilience for device drivers. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 41–50.

[5] JULA, H., AND CANDEA, G. A scalable, sound, eventually-complete algorithm for deadlock immunity. 119–136.

[6] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not. 43*, 3 (2008), 329–339.

[7] MURPHY, B. Automating software failure reporting. *Queue 2*, 8 (2004), 42–48.

[8] RYZHYK, L., CHUBB, P., KUZ, I., AND HEISER, G. Dingo: taming device drivers. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 275–288.

[9] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997), 391–411.

[10] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Trans. Comput. Syst. 24*, 4 (2006), 333–360.

[11] SWIFT, M. M., MARTIN, S., LEVY, H. M., AND EGGERS, S. J. Nooks: an architecture for reliable device drivers. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), ACM, pp. 102–107.

[12] WITKOWSKI, T., BLANC, N., KROENING, D., AND WEISSENBACHER, G. Model checking concurrent linux device drivers. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ACM, pp. 501–504.