Asim Kadav

Research Statement

Address: 1210, W. Dayton St., Madison, WI 53705.
Phone: (608) 698 5025
Email: kadav@cs.wisc.edu
WWW: http://cs.wisc.edu/~kadav

I am interested in improving support for device access in operating systems. With the proliferation of USB based devices, users are attaching an ever growing, changing and evolving set of devices to make the computers more functional and interesting. Furthermore, the introduction of new forms of I/O, such as touchscreens, accelerometers, and new interfaces, such as thunderbolt, have further increased the required OS abstractions. These devices are also being accessed in different environments such as through hypervisors or remotely through a cloud. Thus, not only are number of devices growing but the requirements placed on supporting them are also growing. My goal is to make device access fast, reliable and provide the required abstractions to use these devices. This aligns with my long term goal of making computing more useful and reliable at reduced costs.

My dissertation looks at three related problems to improve device-access: 1) Deeper understanding of the breadth of driver code, 2) Reliability of device drivers against device failures, and 3) Improving device downtime latency by introducing checkpoint-restore in modern drivers.

Understanding Modern Device Drivers

In order to improve device access, I looked at the range of problems that affect device drivers. However, I realized that limited information about the breadth of drivers is available. For example, most research projects consider only a small set of PCI devices, and then generalize to all driver classes. However, many devices for consumer PCs are connected over USB, and hence these designs may not apply. As a second example, research in the past decade, identified drivers as the biggest source of bugs and, as a consequence, a primary source of reliability and security problems in Linux. However, a recent study (Palix et al., ASPLOS '11) found that this bug density of drivers is no longer different than rest of the kernel, indicating that perhaps many of the reliability problems affecting the driver-kernel interface have been addressed.

The implications of broad information extend beyond the applicability of existing research. They help guide the fundamental type of research that we should perform as a community. Motivated by this opportunity for drivers, I performed the largest-ever study [3] of Linux driver source code. I developed a set of static analysis tools to analyze 5.4 million lines of driver code across various axes. Broadly, my study looks at three aspects of driver code (i) what are the characteristics of driver code functionality and how applicable is driver research to all drivers? (ii) how do drivers interact with the kernel, devices, and buses, and how can we achieve driver code standardization? (iii) are there similarities that can be abstracted into libraries to reduce driver size and complexity?

My study produced several interesting results with far reaching implications and I'll discuss the top three. First, most of the driver code is dedicated to driver initialization and cleanup (about 36%) and only 23% is dedicated to handling I/O and interrupts. These results indicate that efforts at reducing the complexity of drivers should not only focus on request handling, but also on better mechanisms for initialization. Second, I found that many assumptions made by driver research do not apply to all drivers. For example, drivers are categorized into different classes based on their functionality such as network, disk etc. However, I found at least 44% of drivers have functionality outside this class definition. Hence, existing driver recovery techniques will fail to correctly recover from failures for 44% of the devices. Finally, from our similarity study, I found 8% (or about 500 KLOC) of all driver code is substantially similar to code elsewhere that adds unnecessary complexity to drivers. This code can be removed with new abstractions, libraries or programing techniques.

My study has provided the community with a better understanding of driver behavior, of the applicability of existing research to these drivers, and of the opportunities that exist for future research such as driver synthesis and remote driver execution.

Tolerating Hardware Device Failures in Software

Prior works, have focused mostly on examining the implications of the driver-kernel interface on the reliability of drivers. I extend the scope of driver reliability research by examining the driver-hardware interface. In examining this interface, my aim is to answer the following question: "How do drivers behave when hardware is faulty, and what can I do to improve it?"

Studies on servers from Microsoft indicate that transient hardware failures are a common cause of unplanned outages. When drivers use unverified hardware inputs, it may crash the system. I define these bugs as *hardware dependence bugs*, which are a new class of bugs representing instances where drivers make incorrect assumptions about hardware behavior. My research was the *first* research consideration to consider the implications of hardware unreliability problems on drivers.

More importantly, the Microsoft study also showed that by augmenting existing drivers with fault-tolerant code, the system is able to tolerate over 60% of the hardware related faults. The existing approach to this problem requires time consuming effort by the developer to develop well-written hardware fault tolerant code. Various OS and device vendors layout guidelines that require developers to ensure that drivers always validate all hardware inputs, ensure device operations finish in finite time, report all device errors and provide provisions for recovery after failures.

To solve the above problem, I developed a system called *Carburizer* that automatically implements the above vendor recommendations [2]. It consists of three components. First, a static analysis component detects misuse of device inputs in critical control and data paths. Second, a code patching component automatically fixes these bugs by adding suitable checks. This component also generates logging code for places where drivers detect failures but does not report them. Finally, a runtime watchdog component fixes problems that cannot be fixed statically such as interrupt and timing bugs. It also provides a recovery service that restores the driver to a last working state when a hardware dependence check fails.

Carburizer's static analysis is fast, scanning upwards of 3000 drivers in less than 30 mins while also generating hardened binary drivers. Using Carburizer, I was able to find 992 hardware dependance bugs in the Linux 2.6.18.8 driver tree. Additionally, I found approximately 1100 cases where the driver was missing error reporting information about device failures. Carburizer hardened driver and runtime imposed almost no performance or CPU overheads.

I presented this system at Linux Plumbers Conference, 2011 to inform Linux kernel developers of these bugs. The system was also featured in a Linux Weekly News Article (Feb 2012), which described this problem, my paper and a list of >1000 bugs from Linux 3.1 generated by Carburizer. I was subsequently contacted by many kernel developers thanking me for my efforts.

Fine-Grained Fault Tolerance Using Device Checkpoints

Carburzier uses the restart-replay approach of recovery which requires logging each and every call to the driver. However, recovery is slow; restart takes considerable time since the driver performs a full probe sequence to determine the type, capabilities and environment for the device. Replay may be incomplete since my driver study showed that 44% of drivers implement non-standard features that cannot be captured for replay. In addition, these systems either require large subsystems that must be kept up-to-date with the the kernel, or require substantial rewriting of drivers.

Support for checkpoint and recovery in drivers can reduce driver downtime and can restore state independent of how a function modifies driver state. However, to date this feature is not possible for drivers because they share state with their devices that is not available through memory. As a result, capturing enough state to restore driver functionality following a failure is difficult. This functionality is often considered to require significant re-engineering of device drivers.

I developed a novel checkpointing mechanism that re-uses existing suspend/resume code in drivers and described a principled way to refactor existing drivers to export the checkpoint-restore interface. I demonstrate its utility by building Fine-Grained Fault Tolerance (FGFT), a system that

provides fine-grained control over the code protected [4]. FGFT executes driver entry points as a transaction and uses software-fault isolation to prevent corruption and detect failures. When a call fails, FGFT discards the software state and restores the device from a checkpoint. Furthermore, FGFT can be integrated with static analysis tools (such as Carburizer), to provide fault tolerance for specific entry points statically or during runtime, when specific inputs occur.

Results from FGFT are encouraging: First, taking a checkpoint is fast, averaging only 20μs. Second, checkpoints remove the need for logging and hence remove the problem of incomplete recovery arising from unique device semantics. Third, the implementation effort of FGFT is small: I added 38 lines of code to the kernel to trap processor exceptions, and found that device checkpoint code can be constructed with little effort from power-management code present in 76% of drivers in common driver classes. Finally, FGFT is able to tolerate a range of driver failures. While I applied checkpoints to fault tolerance, there are more opportunities to use them, such as in OS migration, fast reboot, and persistent operating systems.

Other Research: Apart from my dissertation research, I have worked on broad set of problems at Wisconsin and my internships at Microsoft Research. This includes how to perform live migration of devices directly attached to guest virtual machines [1] and testing of drivers using static analysis and symbolic execution [5]. During my internships, I've looked at storage, including reliability of SSD RAID [6], and how to provide efficient block access to storage in data centers (Under Submission). I also have full time work experience of three years as a developer working on distributed filesystem (GPFS) and databases, which adds breadth to my experience.

Research Style

These projects demonstrate my research style with three characteristics. First, I enjoy *multi-disciplinary* research. For example, many of my projects at Wisconsin have ideas from the PL community. In future, I would like to continue such collaborative research especially in PL and networking areas. Second, my research attempts to solve problems *completely*. For example, Carburizer addressed many aspects of hardware failures (device misuse, failure logging and recovery). Similarly, my driver study looked at the breadth of all drivers as opposed to looking at a small sample, which required developing tools that scale. I believe in taking a holistic view to research. I examine the range of related problems, extract high level commonalities, and design a small number of solutions that attack these commonalities and solve the broad range of problems. Third, my research is *real* and immediately applicable. For example, Carburizer identified many bugs for Linux community. The tool is available for download, and is currently being used in at least one other research project.

Future Research

I intend to extend my existing research on drivers in the short term, and OS/device co-design and I/O virtualization in the long term.

Future Directions in Driver Research: In the short term, I plan to look at problems in drivers that are complimentary to existing research. These include looking at device protocol violations in drivers and re-design of graphics drivers. Device protocol violations occur when drivers incorrectly interact with devices. Often, drivers are written from hardware specifications with poor/missing documentation. These problems are difficult to detect since they require a working device in order to verify that device protocols have been implemented correctly. My goal is to automatically infer device behavior from existing driver code and detect its inconsistent use within drivers using static analysis. I plan to leverage my past work on detecting device failures to find these bugs.

Graphic drivers are among one of the most complex, largest and fastest growing drivers in the kernel. Studies show that up to 40% of Windows Vista crashes in 2007 are exclusively due to graphics drivers. However, modern reliability research has not looked at graphics drivers for multiple reasons. First, they are large and complex. Hence, many automatic refactoring tools or thorough testing tools such as symbolic execution choke on graphics drivers. Second, the graphics subsys-

tem is complex, consisting of multiple libraries and drivers, often consisting of code from different vendors and licenses. My goal is to re-design the graphics architecture for reliability comprising of clean, coarse grained interfaces that reduce complexity and allow driver modules with different licenses to co-exist. I plan to use results from my driver study to separate OS and hardware specific code and identify opportunities to extend this design to other interfaces such as SCSI.

Co-designing device and OS abstractions: In the long term, I plan to develop device/OS abstractions for specific goals such as energy efficiency. As an example, DRAM is a significant consumer of power in servers. Unlike processors, which can be partially turned off with clock gating or turning off cores, DRAM must be powered off in its entirety. Partially turning off DRAM to reduce power is challenging because the OS uses all available space as file cache. Furthermore, interleaving of memory pages inside DRAM hinders powering off specific banks inside DRAM. I plan to co-design low power modes with OS page management policies. Specifically, I plan to co-design how physical pages internally map to the use of OS pages, and how to turn off ranks within DRAM as pages are evicted. I plan to explore (1) How can existing DRAM low power modes be extended to partially turning off DRAM? (2) How to consolidate memory given the effects of memory fragmentation and limited free memory due to file cache? and (3) How to use this a new low powered memory tier transparently and reliably in the OS?

Re-architecting I/O in virtualization era: With the onset of cloud computing and increase in personal devices, I look forward to solve the challenges within I/O virtualization related to reliability, security and flexibility of device access. As an example, the following trends require a rethink of device architecture in virtualized environments. First, devices other than CPU and memory are increasingly being accessed remotely (such as Amazon's EBS). Second, micro-servers (such as single fabric computers) that perform all their I/O over the network are being used. Third, there is an increase in the processing power of devices. I plan to look at the existing I/O stack and (1) Design remote I/O communication that removes inefficiencies of current fine-grained access, (2) Improve driver code standardization by removing device specific code from the kernel, (3) Support access to proprietary device features using higher level communication mechanisms and (4) Support low latency device access for applications, such as giving a database direct access to a disk.

Finally, I hope to find an environment as exciting as Wisconsin to continue to engage in relevant, high-impact research and foster long term collaborations.

Publications

1. Asim Kadav and Michael M. Swift. Live Migration of Direct-Access Devices. In *Proceedings of ACM SIGOPS Operating Systems Review, "Best Papers from VEE and WIOV, 2008"* **(OSR '09)**, Volume 43, Issue 3, July 2009.
2. Asim Kadav, Matthew J. Renzelmann, Michael M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* **(SOSP '09)**, Big Sky, MT, October 2009.
3. Asim Kadav and Michael M. Swift. Understanding Modern Device Drivers. In *Proceedings of the 17th ACM Conference on Architectural Support for Programming Languages and Operating Systems* **(ASPLOS '12)**, London, UK, March 2012.
4. Asim Kadav, Matthew J. Renzelmann and Michael M. Swift. Fine-Grained Fault Tolerance using Device Checkpoints. In *Proceedings of the 18th ACM Conference on Architectural Support for Programming Languages and Operating Systems* **(ASPLOS '13)**, Houston, TX, March 2013.
5. Matthew J. Renzelmann, Asim Kadav, Michael M. Swift. SymDrive: Testing Drivers Without Devices, In *Proceedings of Symposium on Operating Systems Design and Implementation* **(OSDI '12)**, Hollywood, CA, October 2012.
6. Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD Reliability. In *Proceedings of the Fifth ACM European Conference on Computer Systems.* **(Eurosys '10)**, Paris, France, April 2010.