Computer Sciences Department

Live Update for Device Drivers

Michael M. Swift
Damien Martin-Guillerez
Muthukarappan Annamalai
Brian N. Bershad
Henry M. Levy

Technical Report #1634

March 2008

UNIVERSITY OF WISCONSIN MADISON

Live Update for Device Drivers

Michael M. Swift*, Damien Martin-Guillerez, Muthukaruppan Annamalai,
Brian N. Bershad and Henry M. Levy

Computer Sciences Department
University of Wisconsin–Madison
swift@cs.wisc.edu, dmartin@irisa.fr, muthu@cs.washington.edu,
berhad@google.com, levy@cs.washington.edu

Abstract

As commodity operating systems become more reliable and fault-tolerant, the availability of a system will be determined not by when it crashes, but instead by when it must be shutdown and rebooted due to software maintenance [12, 21]. While many system components can be upgraded on-line, critical low-level components, such as device drivers and other kernel extensions, cannot be updated without rebooting the entire operating system.

In this paper, we present *Live Update*, a mechanism that allows device drivers to be updated without rebooting the system. Unlike other on-line update mechanisms, our system supports existing drivers "as is". Thus, thousands of existing device drivers can be updated transparently. In experiments we show that Live Update can upgrade existing drivers without rebooting and that the system imposes very little performance overhead.

1 Introduction

Commodity operating systems, such as Linux and Windows, are becoming more reliable. Each successive release provides more functionality with fewer crashes than the previous [21, 20, 5]. This reliability trend is enabled by a combination of natural maturity and recent technologies that eliminate bugs automatically [7, 2] or that tolerate failure when it occurs [29, 9, 18, 25]. However, even with these improvements, operating systems require *scheduled downtime* for maintenance of certain components [21, 12]. For example, components the OS itself depends on, such as storage drivers, generally cannot be replaced without rebooting. Thus, scheduled downtime promises to be the ultimate limiter of system availability because operating systems cannot replace all running code on-line.

This problem is compounded by the increasing rate of software updates. Previously, manufacturers had to distribute a CD or floppy disk to release an update. Now, Internet services such as Windows Update [19], Apple's Software Update [1], and Red Hat Network [23] greatly lower the cost and complexity of releasing updates. While improving the average quality of running software, frequent software updates may in fact *decrease* the availability of many systems when updates need to reboot the system [26].

Device drivers represent a large portion of this problem because they are both critical to OS operation and frequently updated. For example, Linux IDE storage drivers, needed by the OS for virtual memory swapping, were updated more than 67 times in three years. Furthermore, updating device drivers is inherently risky, because a faulty device driver may prevent the system from booting. To achieve high availability, operating systems must be able to replace critical device drivers without rebooting the entire system.

This paper presents a new mechanism, called *Live Update*, that updates device drivers in place without rebooting the operating system. Our mechanism uses *shadow drivers* [28] to reinitialize the updated driver transparently to the operating system and applications. In addition, the shadow driver transfers the state of the old driver to the new driver automatically, ensuring that driver requests will continue to execute correctly after the update. Live Update improves availability by updating drivers without rebooting the OS, updating drivers without restarting applications that use an updated driver, and rolling back driver updates that do not work.

Updating drivers on-line presents three major challenges. First, there are thousands of existing drivers, many of which will eventually require an update. To make Live Update practical, we need to be able to update these drivers without any work on the part of the drivers' authors. Previous systems for updating modules on-line require that a skilled programmer write code to transfer data between the old and new modules [8, 6, 15, 22]. In contrast, our system uses the pre-existing driver interface to transfer state between driver versions, allowing it to update drivers on-line without driver modifications. Second, an updated device driver may not be compati-

*Work done while at the University of Washington

ble with the running version and could cause applications or the OS to crash. Thus, Live Update must ensure that an update is *safe* and does not contain incompatibilities that could compromise system reliability. Lastly, device drivers can be critical to overall system performance, so the update mechanism must impose little overhead when not actively applying an update.

We implemented Live Update for sound, network, and storage drivers in a version of the Linux operating system. Our results show that Live Update (1) minimally impacts system performance, (2) allows transparent, on-line upgrade between a wide range of driver versions, and (3) can be used without changing existing drivers.

The rest of this paper describes the design, implementation and performance of Live Update. The following section reviews previous approaches to updating code on-line. Section 3 discusses the unique issues that arise when updating device drivers. Section 4 describes the design and implementation of Live Update. Section 5 presents experiments that evaluate the effectiveness and performance of our system, and Section 6 summarizes our work.

2 Related Work

Updating code on-line has long been a goal of high availability systems. Live Update focuses on updating a single OS component type, the device driver. By focusing on driver updates, we can leverage the properties of drivers to transparently resolve a major source of downtime while maintaining a low runtime overhead. Previous approaches differ from our work in terms of the programmer's responsibility and the update mechanisms and granularity. We discuss each of these in turn.

In terms of responsibility, previous on-line update systems typically require a programmer to provide code to facilitate the update. Some systems use compiler support for updating programs [17, 15]. While the program code itself need not change, a programmer must provide a function to translate data formats when they change. Other systems require that updatable code use special calling conventions [8], inherit from an "updatable" base class [16, 22], use an "updating" framework [27], or identify safe update locations in the code [14]. By leveraging the properties of the code being updated, a system can reduce the programmer's role. For example, one system updates event handlers by running old and new versions in parallel until their outputs converge [10]. However, this approach still requires that a programmer write code to transfer the configuration of a handler.

In contrast, our system can update existing, unmodified modules without code specific to an individual module. We leverage the common interfaces of device drivers to automatically capture and transfer their state between versions. By centralizing the code for update in an operating system service, we eliminates the need to modify existing drivers.

In the past, two major mechanisms have been used to update code. The system can launch a copy of a running process with the new code, on the same hardware [13] or on additional hardware [24], and then copy in the state of the existing process. Or, the system can retain running code and redirect callers of an updated procedure to the new version. Redirection has been provided by programming frameworks that incorporate wrapper functions [8, 11, 27], class hierarchies mandating dynamically dispatched functions [16, 22, 6], and compilers that produce only indirect function calls [17, 15]. Live Update similarly redirects callers at updated drivers using wrapper functions. However, Live Update redirects calls both into and *out of* drivers to prevent a driver from calling the OS during an update, thereby concealing the update from the OS itself.

Previous systems have enabled update with varying granularity. Several systems replace single functions or lists of functions on-line [17, 8, 15]. Object-oriented approaches update an entire class [16, 6, 22]. Update mechanisms for modularized systems replace a single module at a time [27]. Because device drivers consist of multiple modules joined by internal interfaces, Live Update replaces entire drivers. This approach allows changes in the interfaces between driver modules, which increases the variety of updates that can be applied on-line.

Partial reboots, which avoid the need to reboot an entire system, have been proposed as a general technique for building highly available fault-tolerant systems [3, 4]. Partial reboots of device drivers avoid a whole-system reboot when a driver fails [29, 28]. Unlike previous work, where partial reboots were used to repair failed components, our system uses partial reboots to update components instead. The system shuts down the current driver and starts the new driver transparently to the driver's callers.

3 Issues

Device drivers have unique characteristics that influence the design of an on-line update system. Drivers are commonly implemented as a set of modules that can be dynamically loaded into the OS kernel. They are organized into classes that share a common programming interface. The common interface allows the kernel and applications to access the device without being aware of the device's specific characteristics. For example, all sound card drivers share a common interface, allowing a sound-playing application to access the sound card without having to manage the card itself.

The need to update existing unmodified driver code raises several design issues. First, the update system must

rely on the existing capabilities of drivers, such as dynamic loading, to initialize the new driver. As well, the update system must be responsible for transferring driver state because there is no code in existing drivers to transfer state between driver versions.

Second, because drivers assume exclusive access to devices and require access to the device to initialize, the old driver must be disabled while the new driver initializes. Thus, the device is unavailable during an update. Any portion of the update process that depends on a device being present (e.g., loading the new driver off a disk when updating the disk driver) must be performed before the old driver is disabled.

Finally, an updated driver may offer substantially different services or support different interfaces. For example, a new version of a driver may remove functions or capabilities provided by the prior version. The update system must be able to detect this situation and disallow the update.

4 Live Update: Design and Implementation

Live Update reflects a systems-oriented approach to updating device drivers on-line. Rather than relying on drivers to update themselves, Live Update is a service that updates drivers on their behalf. Three principles guide the design:

1. *No new code.* It is impractical to write new code for each driver.

2. *Be transparent.* Updates to drivers should be invisible to applications, the operating system, and even drivers.

3. *Safety first.* The system may disallow an update if it is not compatible with running code.

Our system must work with existing drivers because it is too hard, expensive, and error-prone to add on-line update support to drivers. With a systems-oriented approach, a single implementation can be deployed and used immediately with a large number of existing drivers. Thus, we centralize the update code in a single system service. As a common facility, the update code can be thoroughly examined and tested for faults. In contrast, practical experience has shown us that many drivers do not receive such scrutiny, and hence are more likely to fail during an update.

These principles limit the applicability of Live Update. Our solution only applies to drivers that can load and unload dynamically. If the driver cannot, then Live Update cannot replace the driver. Furthermore, our design only applies to drivers that share a common calling interface. A driver that uses a proprietary or ad-hoc interface cannot be automatically updated without additional code specialized to that driver. Finally, if the new driver supports dramatically different capabilities, then the update will fail because the scale of change cannot be hidden from the OS and applications.

Because of these limitations, we consider Live Update to be an optimization and preserve the possibility of falling back to a whole-system reboot if necessary.

4.1 Design Overview

Our system updates a driver by loading the new code into memory, shutting down the old driver, and starting the new driver. During the update, the Live Update system impersonates the driver, ensuring that its unavailability is hidden from applications and the OS. As the new driver starts, the system attaches it to the resources of the old driver and verifies that the new driver is compatible with the old driver. If it is not compatible, then the system can roll back the update. In essence, Live Update replaces the driver code and then reboots the driver.

In a whole-system reboot, the ephemeral state of a driver is lost. When only the driver is rebooted, though, the new driver must preserve the ephemeral state of the old driver, such as configuration parameters, so that application and OS requests can be processed correctly. Furthermore, the reboot must be concealed from applications, as they are generally unprepared to handle driver failures. Rather, they are written with the conventional failure model that drivers and the operating system either fail together or not at all, and therefore do not attempt to handle a driver failure.

Live Update uses *shadow drivers* [28] to safely reboot an updated device driver without impacting applications or the OS. A shadow driver is a kernel agent that is responsible for (1) rebooting a device driver, (2) restoring the driver's state after a reboot, and (3) concealing the driver reboot from the OS and applications. Shadow drivers operate in two modes: passive and active. In *passive* mode, the shadow driver observes all communication between the driver and the kernel. During a driver reboot, the shadow switches to *active* mode, in which it shuts down and restarts the driver while impersonating the driver to the OS. Shadow drivers are normally in passive mode and only switch to active mode during a reboot.

A shadow driver is a "class driver," aware of the interface to the drivers it shadows but *not* of their implementations. A single shadow driver implementation can reboot any driver in the class. Hence, an operating system can leverage a few implementations of shadow drivers to reboot a large number of device drivers. In addition, implementing a shadow driver does not require a detailed understanding of the internals of the drivers it shadows.

Update Steps
1. Capture driver state
2. Load new code
3. Impersonate driver
4. Disable/unload old driver
5. Initialize new driver
6. Transfer driver state
7. Enable new driver

Table 1: **Live Update process for updating a device driver.**

Rather, it requires only an understanding of those drivers' interactions with the kernel.

Table 1 lists the steps of updating a driver. The process begins when a driver is initially loaded and the system starts capturing its state and ends after the driver reboots and the new driver begins processing requests. We now describe each of the update steps in more detail.

4.2 Capturing Driver State

The state of a device driver consists of the state it receives from hardware and the state it receives from the kernel. From hardware, a driver receives persistent data and environmental parameters, such as the speed of a network link, which need not be explicitly preserved when a driver is updated. From the kernel, the driver receives configuration and I/O requests. While configuration changes must persist across an update, completed I/O requests are forgotten by drivers and have little impact on future requests. Hence, they need not be retained after a driver update. As a result, the state of a driver is determined by the history of configuration requests it receives and the requests it is currently processing.

Live Update captures the state of a device driver by observing the communication between the driver and the kernel. When a driver loads, Live Update interposes *wrappers* on all function calls between the driver and the kernel. The wrappers serve two purposes: they provide *taps* that mirror communication to the shadow, and they provide a layer of indirection between a driver and the kernel that allows Live Update to redirect calls to a new driver. We discuss redirection in Section 4.6. From the taps, the shadow records information necessary to unload the old driver and initialize the new driver to a point where it can process requests correctly.

Each function call between the driver and the kernel invokes a tap. The tap first calls the function and then calls the shadow with the function's parameters and result. From this communication, the shadow tracks the kernel objects currently in use by a driver in an *object tracker*. For example, the shadow records the I/O memory regions used by the driver. The shadow also records open connections to the driver and pending requests that

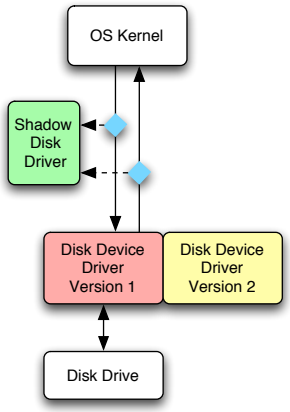

Figure 1: **A shadow driver captures the old driver state by monitoring its communication with the kernel while a new driver version is pre-loaded into memory.**

the driver is processing. To capture the configuration of the driver, the shadow logs requests that set the driver's options or parameters. From the information in the object tracker and log, the shadow driver can restore the driver's internal state after a reboot. A sample shadow driver capturing driver state is shown in Figure 1.

4.3 Loading Updated Drivers

A system operator begins an update by loading the new driver code. As discussed in Section 3, the services of a driver are not available while it is rebooting. To avoid circular dependencies that could arise if updates require the services of the driver being updated, an operator must pre-load the new driver version into memory. As a result, the system can update disk drivers that store their own updates.

Live Update logically replaces entire drivers. When a driver consists of multiple independent modules, though, the system only replaces modules whose memory images change. In addition to updated modules, the system reloads modules that call into updated modules, in order to link them against the new code. As an optimization, Live Update reuses unchanged modules from memory. By replacing the entire driver, the interfaces between a driver's modules can be updated.

Figure 1 shows the new version loaded in memory alongside the existing version. Once the new code is in memory, an operator notifies Live Update to replace the driver. Because both versions are in memory during an update, the system can revert to the old driver if the update fails.

4.4 Impersonating the Driver

When an operator triggers an update, the shadow driver switches to *active* mode. The taps block direct communi-

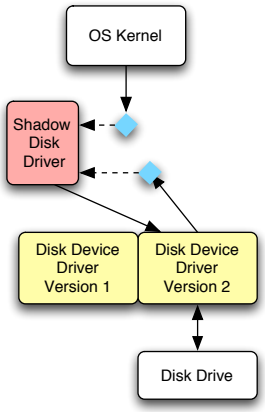

Figure 2: **A shadow driver handling kernel requests during an update .**

cation between the kernel and the driver and instead route all requests to the shadow. The shadow driver acts as the kernel to the driver and as the driver to the kernel, and the details of updating a driver are left to the shadow. Communication in active mode is illustrated in Figure 2.

In active mode, the shadow impersonates the driver by handling requests on the driver's behalf. The shadow may suspend some callers, but because drivers may take significant time to start, the shadow cannot suspend all callers without negatively impacting applications. Depending on the type of request, the shadow itself responds, either using information from the log or by queuing the request and returning to the caller immediately. The shadow driver later submits queued requests to the driver after the reboot completes. Thus, the shadow masks the unavailability of the driver, ensuring that the OS and applications continue to execute correctly.

4.5 Unloading the Old Driver

While impersonating the driver, the shadow begins the driver reboot by shutting down the old driver. The shadow disables the driver, waits for kernel threads to finish using the driver, and then releases the driver's resources on its behalf. Thus, even a driver with bugs in its unload code can be updated safely.

The shadow releases the majority of the kernel objects it tracked. A few objects – those that the kernel uses to call the driver – must be retained in order to conceal the driver's reboot from the kernel. For example, the shadow retains function pointers used by the kernel to call the driver.

In addition to releasing kernel objects, the shadow resets the old driver to its initial state so it can be used for rolling back an update. The shadow copies back the old driver's initial code and data from clean versions saved when the driver was loaded. This process resets the old driver to the same state as the new driver, allowing it to be restarted if the update fails.

4.6 Initializing the New Driver

Once the old driver is unloaded, the shadow driver initializes the new driver with the same sequence of calls that the kernel makes when initializing a driver. By making these calls itself, the shadow conceals the driver's initialization from the kernel proper. As the driver initializes, it calls into the kernel to register and acquire resources. Taps direct these calls to the shadow, which handles the calls on the kernel's behalf by connecting the new driver to the old driver's resources and registration. For example, when a driver calls to register itself with the kernel, the shadow updates the existing driver object in the kernel to reference the new driver instance.

The shadow faces two major challenges in initializing the new driver: identifying and updating the existing kernel objects that the new driver requests, and ensuring that the new driver is compatible with the old driver. We now discuss each of these challenges in detail.

4.6.1 Updating References

When the new driver requests a resource from the kernel or provides a resource to the kernel, the shadow driver must locate the corresponding resource belonging to the old driver. Once located, the shadow updates the resource to reference the new driver and returns it to the driver. The shadow must locate the corresponding resource from the old driver, and then safely update it to reference the new driver.

In most cases, drivers uniquely identify their kernel resources, which allows the shadow to locate the old driver's corresponding resource. For example, drivers provide a unique name, device number, or MAC address when registering with the kernel, and unique memory addresses when requesting I/O regions. The shadow uses the unique identifier and resource type to locate the old driver's resource in the object tracker.

Once located, the shadow updates the kernel resource to reflect differences between the old and new driver. For example, the new driver may provide additional functions or capabilities. Typically, the shadow relies on the indirection provided by wrappers to leave kernel objects unchanged. For example, the shadow updates wrappers to point to the new driver's functions instead of changing function pointers in the kernel. Otherwise, the shadow updates the kernel object directly.

In some cases the driver passes a resource to the kernel without unique identification. The shadow updates these references after the new driver finishes initializing by repeating the calls to the driver that generated the resource in the first place. For example, when the kernel

opens a connection to a sound card driver, the driver returns a table of function pointers with no identifying device name or number. To transfer these function tables to the new driver, the shadow calls into the new driver to re-open every connection. The new driver returns its function table, allowing the shadow to update the connection's table to point to the new driver.

4.6.2 Detecting Incompatibilities

A major problem when updating unmodified code is ensuring that an update is compatible with the running code. If an update is not compatible, then applying it may cause the OS and applications to fail. Incompatibilities arise when a new driver version implements different functions (e.g., higher-performance methods), different interfaces (e.g., a new management interface), or different options for an existing function. Complicating matters, applications may expose incompatibilities long after a new driver starts, for example, when an application attempts to use capabilities that it queried from the old driver. To avoid update-induced failures, the shadow must detect whether the update could cause callers to function incorrectly.

Drivers express their capabilities in two ways. When a driver passes a function table to the kernel, the set of functions in the table describe the abilities of the driver. Drivers also express their capabilities through feature fields. For example, network drivers pass a bit-mask of their features to the kernel, which includes the ability to offload TCP checksumming and do scatter/gather I/O. The shadow considers an update compatible only if a new driver provides at least the same capabilities as the old driver.

The shadow checks compatibility when transferring references to the new driver. For example, when a new driver registers and provides a table of function pointers, the shadow checks whether it implements at least the functions that the old driver provided. If so, the shadow updates the wrapper functions to reference the new driver. The shadow also checks whether the new driver provides at least same interfaces to the kernel as the old driver. For a driver with features that can be queried by applications, the shadow explicitly queries the new driver's features during an update and compares them against the old driver's features.

Live Update can either roll back an incompatible update, or continue with the update and fail any request that would reveal the incompatibility. These options allow an operator to prioritize an important update (for example, one that prevents unauthorized use of the system) over compatibility. Lower-priority incompatible updates can still be applied with a whole-system reboot.

For some driver updates, these rules are overly conservative. For example, if a driver provides a function or capability that is never used, an update need not provide that function. However, it is impossible for the shadow to know if an application will ever *try* to call the function in the future, and hence we use conservative checks.

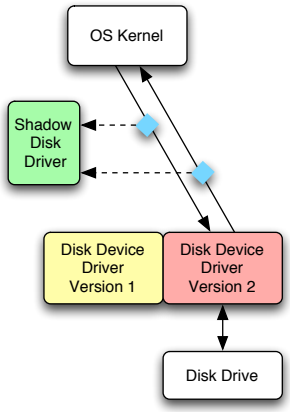

Figure 3: **After an update, the kernel calls into the new driver while the old driver remains in memory in case the update must be rolled back.**

4.7 Transferring Driver State

While previous on-line update systems require applications to transfer their own state between versions, Live Update transfers driver state automatically, without the involvement of driver writers. As previously discussed, a shadow driver captures the state of a driver by monitoring its communication with the kernel. Once the new driver has initialized, the shadow driver transfers the old driver's state into the new driver using the same calls into the driver that it previously monitored. The shadow reopens connections to the driver, replays its log of configuration requests, and resubmits requests that were in-progress when the driver was updated. Once the shadow completes these calls, the new driver can process new requests as the old driver would have prior to updating.

4.8 Enabling the New Driver

After the reboot completes, the shadow switches back to passive mode and restores direct communication, enabling the new driver to begin processing requests. At this time the shadow submits any requests that it queued during the update. An operator may leave the old version in memory if she desires the opportunity to later roll back the update. Or, she may discard the old driver once the update completes. The system state after an update is shown in Figure 3.

4.9 Implementation Summary

Live Update updates device drivers by replacing their code and rebooting the driver. Before an update, it loads

Class	Driver	Device
Sound	**emu10k1**	**SoundBlaster Live! sound card**
	audigy	SoundBlaster Audigy sound card
	es1371	Ensoniq sound card
Network	**e1000**	**Intel Pro/1000 Gigabit Ethernet**
	3c59x	3COM 3c509b 10/100 Ethernet
	pcnet32	AMD PCnet32 10/100 Ethernet
Storage	**ide-disk**	**IDE disk**

Table 2: **The three classes of shadow drivers and the Linux drivers tested. We present results for the boldfaced drivers only, as the others behaved similarly.**

Driver	*Module*	*# of updates*	*# Failed*
emu10k1	emu10k1	10	0
	ac97_codec	14	0
	soundcore	6	0
e1000	e1000	10	1
ide-disk	ide-disk	17	0
	ide-mod	17	0
	ide-probe-mod	17	0

Table 3: **Tested drivers, their modules, the number of updates applied to each module independently, and the number of updates that failed.**

.

driver code into memory to avoid circular dependencies. During an update, shadow drivers hide the driver's unavailability from applications and the OS by handling requests on the driver's behalf.

Live Update calls existing driver interfaces to initialize the new driver and transfer the state of the current driver. As the new driver initializes, the shadow driver connects it to the resources of the old driver. The shadow reassigns kernel structures referencing the old driver to the new driver. Once the new driver has initialized, the shadow driver transfers in the state of the old driver by replaying its log of configuration requests and by reopening connections to the driver. Finally, the shadow dynamically detects whether the new and old drivers are compatible by comparing the capabilities of the new and old drivers. If the new driver is not compatible, either the update or the action that would reveal the incompatibility can be canceled. Essentially, Live Update takes a systems-oriented approach to updating device drivers and provides a centralized service to update unmodified drivers.

5 Evaluation

We implemented Live Update in the Linux 2.4.18 operating system kernel. Based on a set of experiments using existing, unmodified device drivers and applications, our results show that Live Update (1) can apply most released driver updates, (2) can detect when an update is incompatible with the running code and prevent application failures by rolling back the update, and (3) imposes only a minimal performance overhead.

The experiments were run on a 3 GHz Pentium 4 PC with 1 GB of RAM and a single 80 GB, 7200 RPM IDE disk drive. We built and tested three shadow drivers for three device-driver classes: sound card, network interface controller, and IDE storage device. To ensure that Live Update worked consistently across device driver implementations, we tested it on seven different Linux drivers, shown in Table 2. Although we present detailed results for only one driver in each class (*emu10k1*, *e1000*, and *ide-disk*), behavior across all drivers was similar.

Of these three classes of drivers, only IDE storage drivers require a full system reboot in Linux to be updated. However, updating members of the other two classes, sound and network, disrupts applications. Live Update improves the availability of applications using these drivers, because they need not be restarted when a driver is updated.

In the rest of this section, we answer two questions about our Live Update implementation:

1. *Effectiveness.* Can Live Update transparent update existing drivers, and can it detect and roll back incompatible updates?

2. *Performance.* What is the performance overhead of our mechanism during normal operation (i.e., in the absence of updates), and are updates significantly faster than rebooting the whole system?

5.1 Effectiveness

To be effective, the Live Update system must hide updates from the applications and the OS, apply to most driver updates, and be able to detect and handle incompatible updates.

Because Live Update depends on shadow drivers to conceal updates, it shares their concealment abilities. Previous work on shadow drivers [28] demonstrated that shadow drivers could conceal driver reboots from all tested applications in a variety of scenarios. We therefore only examine the latter two requirements: whether Live Update can apply existing driver updates and whether it can detect and roll back incompatible updates.

Applicability to Existing Drivers

We tested whether Live Update can update existing drivers by creating a set of different versions of the drivers. We compiled past versions of each driver, which we obtained either from the Linux kernel source code or

from vendors' web sites. For drivers that consist of multiple modules, we created separate updates for each module and updated the modules independently. Because the driver interface in the Linux kernel changes frequently, we selected only those drivers that compile and run correctly on the the 2.4.18 Linux kernel. Table 3 show the modules comprising each of our tested drivers and the number of updates for each module.

We applied each driver update from oldest to newest while an application used the driver. For the sound driver, we played an mp3 file. For the network drivers, we performed a network file copy, and for the IDE storage driver we compiled a large program. For each update, we recorded whether the driver was successfully updated and whether the application and OS continued to run correctly. If the update process, the application or OS failed, we consider the update a failure.

The results, shown in Table 3, demonstrate that Live Update successfully applied 90 out of 91 updates. In just one case, when updating the *e1000* driver from version 4.1.7 to 4.2.17, the shadow was unable to locate the driver's I/O memory region when the new driver initialized. This is because the new driver used a different kernel API to request the region, which prevented the shadow from finding the region in the object tracker. The new driver failed to initialize, causing Live Update to roll back the update. The file-copy application, however, continued to run correctly using the old driver.

This single failure of Live Update demonstrates the complexity of transferring kernel resources between driver versions. Shadow drivers use a unique identifier to locate the resource of an old driver that corresponds to a new driver's request. For example, interrupt handlers are identified by the interrupt line number. When implementing shadow drivers, we chose unique identifiers for some kernel resources that depend on the APIs for requesting the resource. When a driver changes APIs, the shadow cannot locate the old driver's resources during an update. Ideally, the identifier should be independent of the API used to request the resource. In practice, this is difficult to achieve within the existing Linux kernel API, where there are many resources that can be accessed through several different APIs.

Compatibility Detection

While we found no compatibility problems when applying existing driver updates, compatibility detection is nonetheless critical for ensuring system stability. To test whether the Live Update can detect an incompatible update, we created a new version of each of our three drivers that lacks a capability of the original driver. We created *emu10k1-test*, a version of the *emu10k1* driver that supports fewer audio formats. The shadow detects the sound driver's supported formats by querying the driver's capabilities after it initializes. From the *e1000* driver we created *e1000-test*, a version that removes the ability to do TCP checksumming in hardware. The driver passes a bit-mask of features to the kernel when it registers, so this change should be detected during the update. Finally, we created *ide-disk-test*, a version of *ide-disk* that removes the `ioctl` function. This change should be detected during the update, when the driver registers and provides the kernel with a table of function pointers.

For each test driver, we first evaluate whether compatibility detection is necessary by updating to the test driver with the compatibility checks disabled. Similarly to the previous tests, we ran an application that used the driver at the time of update. Next, we applied the same update with compatibility checking enabled. For each test, we recorded whether the system applies the update and whether the application continued to run correctly.

With compatibility checking disabled, all three drivers updated successfully. The sound application failed, though, because it used a capability of the old driver that the new driver did not support. The disk and network applications continued to execute correctly. The kernel checks the driver's capabilities on every call to disk and network drivers, so it gracefully handles the change in drivers. In contrast, with compatibility checking enabled, Live Update detected all three driver updates as incompatible and rolled back the updates. The applications in these tests continued to run correctly.

These tests demonstrate that compatibility checking is important for drivers whose capabilities are exposed to applications. The sound application checks the driver's capabilities once, when starting, and then assumes the capabilities do not change. In contrast, only the kernel is aware of the network and disk drivers' capabilities. Unlike the sound application, the kernel checks the driver capabilities before every invocation of the driver, and hence the compatibility check during update is not important.

5.2 Performance

Because updates are infrequently applied, an on-line update system must not degrade performance during the intervals between updates. Live Update introduces overhead from its taps, which interpose on every function call between the kernel and drivers, and from the shadow driver, which must track and log driver information. In addition, updates must be applied quickly to achieve high availability.

To evaluate performance cost of Live Update, we produced two OS configurations based on the Linux 2.4.18 kernel:

1. *Linux-Native* is the unmodified Linux kernel.

Device Driver	Application Activity
Sound (emu10k1 driver)	• mp3 player (*zinf*) playing 128kb/s audio • audio recorder (*audacity*) recording from microphone
Network (e1000 driver)	• network send (*netperf*) over TCP/IP • network receive (*netperf*) over TCP/IP
Storage	• compiler (*make/gcc*) compiling 788 C files • database (*mySQL*) processing the *Wisconsin Benchmark*

Table 4: **The applications and workloads used for testing Live Update performance.**

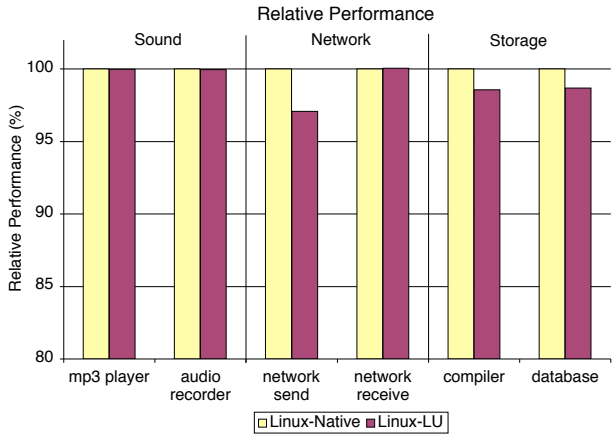

Figure 4: **Comparative application performance on *Linux-LU* relative to *Linux-Native*. The X-axis crosses at 80%.**

2. *Linux-LU* is a version of Linux that includes shadow drivers and Live Update.

We selected a variety of common applications that use our three device driver classes and measured their performance. The application names and behaviors are shown in Table 4.

Different applications have different performance metrics of interest. For the sound driver, we measured the available CPU while the programs ran. For the network driver, throughput is a more useful metric; therefore, we ran the throughput-oriented *network send* and *network receive* benchmarks. For the IDE storage driver, we measured elapsed time to run the applications in Table 4. We repeated all measurements several times and they showed a variation of less than one percent.

Figure 4 shows the performance of *Linux-LU* relative to *Linux-Native*. The figure makes clear that Live Update imposes only a small performance penalty compared to running on *Linux-Native*. Across all six applications, performance on *Linux-LU* averaged 99% of the performance on *Linux-Native*, demonstrating that Live Update imposes little overhead.

The overhead of Live Update can be explained in terms of frequency of communication between the driver and the kernel. On each driver-kernel call, the shadow driver may have to log a request or track an object. For example, the kernel calls the driver approximately 1000 times per second when running *audio recorder*, each of which causes the shadow to update its log. For the most disk-intensive of the IDE storage applications, the *database* benchmark, the kernel and driver interact only 290 times per second. On the other hand, the *network send* benchmark transmits 45,000 packets per second, causing 45,000 packets to be tracked. While throughput decreases only slightly, the additional work increases the CPU overhead per packet by 34%, from on 3μs on *Linux-Native* to 4μs on *Linux-LU*.

Another important aspect of performance is the duration of time when the driver is not available during an update. For each of our three drivers, we measured the delay from when the shadow starts impersonating the device driver until it enables the new driver. The delay for the *e1000* and *ide-disk* drivers was approximately 5 seconds, almost all of which is due to the time the driver spends probing hardware[1]. The *emu10k1* updates much faster and is unavailable for only one-tenth of a second. In contrast, rebooting the entire system can take minutes when including the time to restart applications.

In this section we showed that Live Update could improve availability by updating device drivers without rebooting the OS. For drivers that the OS can replace online, such as sound and network drivers, Live Update was able to update the driver without impacting running applications. We also showed that Live Update was able to automatically roll back updates that failed due to compatibility problems. Furthermore, the overall performance impact of Live Update during normal operation is negligible, suggesting that it could be used across a wide range of applications and environments where high availability is important.

6 Conclusions

While operating systems themselves are becoming more reliable, their availability will ultimately be limited by scheduled maintenance required to update system software. In this paper we presented a system for updating device drivers on-line that allows critical system drivers, such as storage and networking drivers, to be replaced without impacting the operating system and applications, or more importantly, availability.

Live Update leverages shadow drivers [28] to update driver code in-place with no changes to the driver itself. The system loads the new driver code, reboots the driver, and transfers kernel references from the old driver to the new driver. To ensure that applying an update will not re-

[1]The *e1000* driver is particularly slow at recovery. The other network drivers we tested recovered in less than a second.

duce reliability due to changes in the driver, Live Update checks the new driver for compatibility, and can rollback incompatible updates. In testing, we found that Live Update could apply 99% of our existing driver updates and had almost no impact on performance.

References

[1] Apple Computer. Software Update. Available at `http://www.apple.com/support`.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposim on Principles of Programming Languages*, pages 1–3, Jan. 2002.

[3] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small, recursively restartable system. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks*, pages 605–614, June 2002.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[5] D. H. Brown Associates. 2003 Linux Function Review, May 2003. Available at `http://www.dhbrown.com/dhbrown/03_Linux.cfm`.

[6] D. Duggan. Type-based hot swapping of running modules. Technical Report SIT CS Report 2001-7, Stevens Institute of Technology, Castle Point on the Hudson, Oct. 2001.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Dec. 2000.

[8] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 470–476, Oct. 1976.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On- Demand IT Infrastructure*, Oct. 2004.

[10] M. Gagliardi, R. Rajkumar, and L. Sha. Designing for evolvability: Building blocks for evolvable real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 100–109, June 1996.

[11] H. Goullon, R. Isle, and K.-P. Löhr. Dynamic restructuring in an experimental operating system. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 295–304, May 1978.

[12] J. Gray. Why do computers stop and what can be done about it? Technical Report 85-7, Tandem Computers, June 1985.

[13] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Software–Practice and Experience*, 23(9):949–94, Sept. 1993.

[14] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, May 1996.

[15] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 13–23, June 2001.

[16] G. Hjálmtýsson and R. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 65–76, June 1998.

[17] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Apr. 1983.

[18] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gö. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[19] Microsoft Corporation. Windows Update. Available at `http://www.windowsupdate.com`.

[20] Microsoft Corporation. Windows Server 2003 reliability innovations set high standard, Feb. 2003. Available at `http://www.microsoft.com/presspass/press/2003/feb03/02-05ReliabilityPR.asp`.

[21] B. Murphy. Fault tolerance in this high availability world. Talk given at Stanford University and University of California at Berkeley. Available at `http://research.microsoft.com/users/bmurphy/FaultTolerance.htm`, Oct. 2000.

[22] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating for Java software. In *Proceedings of the IEEE International Conference on Software Maintenance 2002*, pages 649–658, Oct. 2002.

[23] Red Hat Corporation. Red Hat Network. Available at `http://rhn.redhat.com`.

[24] M. E. Segal and O. Frieder. On-the-fly program

modification: Systems for dynamic updating. *IEEE Software*, 2(10):53–65, Mar. 1993.

[25] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Oct. 1996.

[26] Smart Computing Editors. Singing the update blues? *Smart Computing*, 15(3):96–97, Mar. 2004.

[27] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 141–154, June 2003.

[28] M. M. Swift, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[29] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), Feb. 2005.