Live Updating Operating Systems Using Virtualization *

Haibo Chen,Rong Chen,Fengzhe Zhang,Binyu Zang
Parallel Processing Institute, Fudan University
{hbchen,chenrong,fzzhang,byzang}@fudan.edu.cn

Pen-Chung Yew
Department of Computer Science and Engineering,
University of Minnesota at Twin-Cities
yew@cs.umn.edu

Abstract

Many critical IT infrastructures require non-disruptive operations. However, the operating systems thereon are far from perfect that patches and upgrades are frequently applied, in order to close vulnerabilities, add new features and enhance performance. To mitigate the loss of availability, such operating systems need to provide features such as live update through which patches and upgrades can be applied without having to stop and reboot the operating system. Unfortunately, most current live updating approaches cannot be easily applied to existing operating systems: some are tightly bound to specific design approaches (e.g. object-oriented); others can only be used under particular circumstances (e.g. quiescence states).

In this paper, we propose using virtualization to provide the live update capability. The proposed approach allows a broad range of patches and upgrades to be applied at any time without the requirement of a quiescence state. Moreover, such approach shares good portability for its OS-transparency and is suitable for inclusion in general virtualization systems. We present a working prototype, LUCOS, which supports live update capability on Linux running on Xen virtual machine monitor. To demonstrate the applicability of our approach, we use real-life kernel patches from Linux kernel 2.6.10 to Linux kernel 2.6.11, and apply some of those kernel patches on the fly. Performance measurements show that our implementation incurs negligible performance overhead: a less than 1% performance degradation compared to a Xen-Linux. The time to apply a patch is also very minimal.

Categories and Subject Descriptors

K.6.3 [*Management of Computing and Information Systems*]: Software Management—Software maintenance; D.4.5 [*Operating Systems*]: Reliability

General Terms

Reliability, Management, Design, Experimentation

* This research was funded by China National 973 Plan under grant numbered 2005CB321905.

VEE'06 June 14–16, 2006, Ottawa, Ontario,Canada.

Keywords

Live Update, Virtualization, Operating System, Availability

1. Introduction

Patches and upgrades are a part of everyday life for a contemporary operating system. Such patches and upgrades are frequently applied in order to plug security holes, add new features and enhance performance. Unfortunately, this process usually requires stopping and restarting a running operating system, which could constitute a major source of its loss of availability. However, for some long running and mission-critical systems, any such disruption could be expensive and intolerable [1]. They have to keep all tasks running all the time, otherwise, risk dire consequences. Therefore, features such as the live update capability [2] have become increasingly important, because it could minimize the planned and unplanned downtime in order to diminish the loss of availability.

Most modern operating systems are large and complex. To live update such operating systems safely, several requirements are identified in [3, 4, 5]. First, updatable units in an operating system need to be easily defined. For an operating system using an object-oriented approach such as K42, an object is a natural updatable unit. Second, a quiescent state or a safe point [6] needs to be detected or enforced before a dynamic patch could be applied. Otherwise, the operating system may result in an inconsistent state. This necessitates an efficient way to track the states of the operating system, for example, using a reference counter to track the number of threads executing in an updatable unit. Finally, an effective approach is required to redirect invocations from the original unit to the newly updated unit after a dynamic patch is applied.

However, most existing operating systems are not designed with a live update capability in mind. First, they are usually implemented using non-object-oriented approaches. Hence, function calls are often made directly rather than going through an indirection table, making it difficult to redirect function calls. Moreover, they often lack well-defined boundaries among various components, preventing component-level live updates. Second, they usually lack the mechanism that supports safe points detection (e.g. reference count). It makes a quiescent state detection either very time consuming or simply impractical. Furthermore, it is very rare for hot spots in an operating system to enter a quiescent state in which live updates can be safely applied. Examples include network modules in a web server and a root file system module. A network module is always busy receiving and sending packets, and a root file system module cannot be unmounted while the operating system is still running. Under such circumstances, emergency patches and updates need to be indefinitely postponed, exposing the whole system to possible attacks or corruption. Finally, even if such a safe state could be reached and detected, due to the fact that the update process is executing inside the operating system, it may trigger an execution of the code in the patch program and result in a dead lock

situation or an inconsistent state. For example, a live update to an interrupt handler may trigger the interrupt and brings the operating system into an undefined state.

Being aware of the above problems, we propose using virtualization as a way to support live updates on existing operating systems. We argue that system virtualization [7], recently a popular technique for many applications, provides the operating system with a seamless capability to support live updates, thus reducing downtime and improving availability.

By running the operating system on a high performance virtual machine, it is convenient and natural for the virtual machine monitor (VMM) to modify the state of the operating system without having to stop and reboot the operating system. We apply live updates at the function level rather than at the component level because it is often impossible to unambiguously partition the whole system into disjoint components. Given that a quiescent state may not even exist in some functions, we eliminate this requirement and instead allow live updates at any time. If a live update changes data, we keep different versions of the data. It is the responsibility of VMM to invoke the state transfer function that maintains the coherence of different versions.

We have built a working prototype, named LUCOS, to provide live update capability to Linux running on Xen [8], a popular open-source VMM. According to our performance measurements, negligible overhead is incurred in such an implementation. We show that several real-life Linux kernel patches could be successfully applied on the fly without the need for a reboot.

The rest of the paper is organized as follows: section 2 presents a brief introduction on system virtualization and its application to provide live update capability. Section 3 describes the overall design of our framework. Section 4 focuses on the detailed design and implementation issues. Section 5 presents some experimental results for Linux on Xen. Section 6 discusses the related work. We close this paper with a brief conclusion.

2. Live Update Using Virtualization

2.1 System Virtualization

In the past few years, virtualization on PC-hardware has been extensively studied, and many systems and innovations have emerged. In general, two trends seem to have dominated the development of system virtualization [9]: *full system virtualization*, where a virtual machine is deployed as a complete replica of the underlying hardware; and *para-virtualization*, where the operating system is modified to support virtualization with a lower performance penalty. Examples of full system virtualization include VMware [10] and Virtual server [11], while Denali [12, 13], User Mode Linux [14], and Xen [8] are some of the typical para-virtualization systems.

Due to the unfriendliness of the IA-32 architecture to virtualization [1], full system virtualization could cause significant performance penalty. Therefore, some research groups advocate the use of para-virtualization by allowing the virtual machine to be close, but not identical, to the underlying hardware. Through exposing some hardware interface to the operating system, para-virtualization significantly reduces the performance penalty, though some modification to the operating system is required.

The Denali isolation kernel allows untrusted services to run in isolated domains. Operating systems must first be ported to the Denali architecture, which is a modified version of x86 with an enhanced virtualizability and scalability. Xen is an x86 VMM developed at the University of Cambridge Computer Laboratory. It is released under the GNU General Public License. A broad range of operating systems have been or will be ported to run on Xen. They include Linux, Windows XP, FreeBSD, NetBSD and Plan 9.

[1] Intel and AMD have announced their plan of hardware enhancement to ease the implementation of full system virtualization, namely Vanderpool and Pacifica, and their commercial products are recently available.

2.2 Applying Virtualization to Live Update

"Any problem in computer science can be solved with another level of indirection." David Wheeler in Butler Lampson's 1992 ACM Turing Award speech.

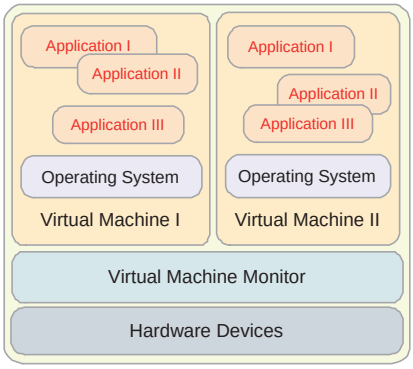

Figure 1. General structure of virtualization.

As depicted in Figure 1, virtualization provides an additional layer between the running operating system and the underlying hardware. The software layer, which is referred to as the virtual machine monitor (VMM), manages the hardware resources and exporting them to the operating systems running on them. As a result, the VMM is in full control of the state and the execution of the operating systems. Thus, it is convenient for the VMM to track and change any state of the operating systems without the need for a reboot.

A live update request is trapped to the VMM that carries out the process of live updates. This method offers several special benefits. First, because any trap from the operating system to the VMM is synchronized and blocked, the operating system is totally inactive during the live update until the trap has completed. This feature ensures the atomicity of a live update.

Second, using the VMM to perform live updates can easily eliminate the requirement of a quiescent state and allow patches to be applied at any time. When there is any change to the data structures in the operating system, we allow the co-existence of both old and new versions. Both versions of the data structures will be write protected by the VMM. If any attempt is made to change either version, a write protection will cause an execution transfer from the operating system to the VMM. A data transfer function is then called in the VMM to ensure the coherence between the old and new versions. When all threads referring the old versions of the data structures have left the old versions of functions, the synchronization can be safely terminated since the kernel is completely using the updated functions and data structures afterwards.

Finally, since the program that applies live update is executed in the VMM, it will not bring unnecessary state change to the trapped operating system. Therefore, it's natural to guarantee the consistence of the operating system when a live update is in progress. In contrast, live updates purely by an operating system itself may cause unexpected changes and bring the system into an unknown state. As an illustration, for patches that update the data structures used in interrupt handlers, write protections in the interrupt handlers will incur double faults and hang the operating system.

3. The Framework of LUCOS

This section presents the general framework of LUCOS. Specifics regarding its implementation are detailed in section 4. We begin

with the high-level design issues that have driven our work, followed by a classification of live updates to operating systems. The final part of this section gives an overview on the architecture of LUCOS.

3.1 Design Issues

To support live updates to a running operating system, LOCUS follows several design principles:

- **OS-Transparency and OS-Neutral:** To avoid disrupting services on a running operating system, any change to the operating system should not necessitate an operating system reboot. Fortunately, most existing operating systems provide some means to extend their functionalities on the fly (e.g., Linux loader kernel module), eliminating the need for reboots. LUCOS takes advantage of this capability through making both the update manager in the target operating system and the dynamic patches in the form of loadable kernel modules, so no modification to the operating system kernels is required. Further, the support for live updates in the VMM is OS-neutral, allowing good portability and easy inclusion in a general-purpose VMM.
- **Flexibility:** LUCOS allows live updating an operating system at the granularity of functions. It also permits updates to both code and data structures, even dynamically adding and removing single instance or multiple instances of data structures. Furthermore, demanding a quiescent state is no longer imperative. Updates are allowed to be performed at any time, even when the code to be updated is still active.
- **Safety and Maintainability:** Any update to the operating system should be transactional to avoid corrupting the whole system. If an error occurs during the update process, the system should be able to roll back any change already made on it. LUCOS also allows any previously committed updates to be rolled back.
- **Correctness:** For simplicity, LUCOS neither verifies nor validates the input patch files, but assume its trustability and correctness. The construction of a patch program is decoupled from the generation of the corresponding patch files, leaving the verification of the program to developers and testers. However, LUCOS allows rolling back problematic patches or patching the same update units more than once.

To render our implementation simple and practical, several design decisions are made as follows:

- **Patches to Data Structures:** To ensure the correctness of the system, it is required that if any change is made to a data structure, any code that manipulates any instance of the data structure should also be updated accordingly.
- **Patches to Function Prototypes:** Since LUCOS mainly changes callee functions and keeps caller functions intact, it's difficult to deal with the case where the function prototype is changed. This problem is solved by requiring that if there is any change to a function's prototype, its callers are recursively chosen as the candidates for live updates. This requirement holds even when patching inlined functions since they have no function body in the binary code.
- **Patches to Initialization Code and Data:** There is usually a large number of initialization code and data in the operating system. They are executed only once during the booting process. Their memory space is freed after the startup phase of the operating system. It is difficult to support live updates to such code and data directly because it is impossible to update those data retroactively. However, such updates usually need to take effective only when the operating system is restarted next time. There is no need to update such code and data immediately to the running operating system. If it is necessary to do so, the effect of such updates can be achieved by providing suitable state transfer functions that appropriately modify the state committed by the initialization code and data.
- **Patches to Scope Information:** The scope information (e.g. static, export) of a function restricts the function's accesses to some specific scope. Such a patch generally has no effect on the runtime behavior of the operating system kernel. However, some may affect the access control rules of the kernel. For instance, the EXPORT_SYMBOL macro in Linux will make a symbol available for use in kernel modules. Therefore, we only focus on patches that explicitly affect the access control rules of the kernel. For patches that change the access rules, they are applied by executing proper state transfer functions. For example, exporting a symbol can be done by adding the symbol into the runtime symbol table of the kernel.

3.2 A Classification of Live Updates

Generally speaking, there could be two types of patches to the operating system: updates affecting only code, and updates affecting both code and data. Because any change to the data structures requires corresponding changes to the code manipulating them, the type of live updates that change only data is not allowed in our system. To facilitate the implementation, a refined classification is made based on the semantic equivalence of a patch program.

Semantic Equivalence: For patches that affect only code, it should be noticed that if the semantic of the patch code is changed when modifying global variables, it should be classified as the second type. As illustrated in the following code fragment, the semantic of the function *foo* is changed since it fixes the deadlock of *demo_lock* by unlocking the demo_lock when the condition doesn't hold. However, the requesting threads to the function *foo* could be deadlocked even after the patch has been applied. Therefore, a callback function that unlocks the *demo_lock* should be called each time a thread leaves the function *foo*.

```
spinlock_t demo_lock = SPIN_LOCK_UNLOCKED;
void foo(void){...;
    spin_lock(&demo_lock);
    ... ;
    if(condition){return;}
    ...;
    spin_unlock(&demo_lock);
}
```

Example code 1: a buggy function with a potential for deadlocks.

```
spinlock_t demo_lock = SPIN_LOCK_UNLOCKED;
void foo_patch(void){
    ...;
    spin_lock(&demo_lock);
    ...;
    if(condition){
       spin_unlock(&demo_lock);
        return;
    }
    ...;
    spin_unlock(&demo_lock);
}
```

Example code 2: a patch function to fix the deadlock problem.

```
void state_transfer(void){
   if(spin_is_locked(&demo_lock))
       spin_unlock(&demo_lock);
}
```

Example code 3: a callback function to recover from a deadlocked situation.

Our final classification of live updates is shown as follows:

- Only code is modified, while existing data structures remain unchanged during the update. The patch code may introduce new local or global variables and data structures, but should maintain the semantic equivalence with the original code.
- Updates that affect both code and data structures in an operating system. The patched data structures can be global, single-instance data, or multiple-instance data. For the later case, as operating systems usually organize all instances of a data structure in some ways, e.g. linking them as a list, locating them can be achieved by iterating all instances.

3.3 Architecture of LUCOS

From a hierarchical viewpoint, LUCOS consists of three major components (see Figure 2). Our design follows a layered approach. The control logic for users is separated from the update logic in the operating system and the VMM, resulting in a clear interface and good portability.

3.3.1 Control Interface

To make the live update system easy to use, a user-friendly control interface is indispensable. It should be easy for users to apply live updates and to rollback existing patches. The users should also have some way of knowing which patches have already been applied. The control interface lies on top of the operating system in the form of a user application. However, only authorized users (e.g. administrators) are allowed to use it. It has three options available.

- **query:** show detailed information of applied patches.
- **patch:** apply a new patch.
- **rollback:** rollback a committed patch.

3.3.2 Update Manager

Since we may need to apply or to rollback patches frequently, it is desirable to manage all these operations in a uniform way and to avoid possible errors in the process. For example, rolling back a non-existent patch should be caught and disallowed, or it may corrupt the operating system. Moreover, there may be cases in which a user patches the same function more than once, so different versions of the patch to the same function must be properly managed. Finally, to allow a patch that involves multiple functions, there should be a mechanism to allow all functions within the patch be grouped and committed together.

The update manager serves as an agent or a proxy between the control interface and the update server. It is in the form of a loadable kernel module in the operating system. It provides the following services:

- receive patch commands from the control interface and verify their legitimacy.
- negotiate with the update server to process live update requests.
- manage the committed patches and coordinate different versions of the patches to the same function.

A live update command is sent to the update server in the form of a hypercall [8]. A hypercall is a synchronous software trap that carries out the control transfer from the operating system to the VMM on which it runs. It is analogous to a system call in the conventional operating system. The synchronous nature of the hypercall ensures that the operating system is inactive in its entirety during the live update process in VMM.

3.3.3 Update Server

The update server lies in the VMM and carries out the real job of live updating code and data. It receives all of the necessary information from the update manager. The function of the update server includes redirecting function calls, setting up necessary data structures to maintain the coherence among different versions of the data, invoking the state transfer functions.

The update server is composed of several hypercall handlers that service the corresponding live update requests. The update server is responsible for guaranteeing the coherence between the original and the new data if the live update changes the data structures. Upon receiving a notification that either version of the data has been modified, the update server invokes the corresponding state transfer function supplied by the update manager to maintain coherence.

4. Detailed Design and Implementation

We have implemented a working prototype on Linux 2.6.10 running on Xen-2.0.5. The hardware platform is the Intel x86 architecture (a ”P6” or newer processor). We chose Linux and Xen because of their broad acceptance and the availability of their open-source code. Xen supports or is to support a broad range of operating systems that include Windows XP, FreeBSD, NetBSD and plan9. Linux also has a good support of kernel extensions in the form of loadable kernel modules.

The following subsections discuss the detailed design and the implementation of LUCOS. First, we describe how to define and to generate patch files for live updates. Then, we present in detail how to perform and to rollback a live update. Finally, we discuss some open issues related to the implementation.

4.1 Patch Construction

The source code of the patches used in our experiments is obtained from real-life Linux kernel modifications made by kernel developers. Linux kernel developers have announced numerous patches to fix security holes, add new features and enhance performance. They are in the form of static patches, which are generated by differing the old version of Linux with the updated version.

Applying a live update to a Linux kernel generally involves four steps: (1) analyze the static patch, (2) write a source file for the live update patch, (3) generate an executable binary for the patch file, (4) inject the binary patch file and apply the live update. Here, we focus on defining a simple and powerful input file format and giving some guidelines to write a LUCOS patch. Also, we supply some helper functions to ease the construction of LUCOS patches.

4.1.1 LUCOS Patch Files

Linux allows a dynamic injection of kernel code in the form of a loadable kernel module, which defines the basic format for the patch files. To make our system powerful enough to handle various complex kernel patches, some additional functions and fields are added to the patch files.

A kernel patch file usually consists of updates on several functions and data structures. As LUCOS allows co-existence of both the old and new versions of the data structures, data transformation functions are needed to maintain their coherence. In some cases, a one-to-one correspondence between the data structures is not enough for complex patches. For example, adding new fields or merging data structures may require a multiple-to-one conversion,

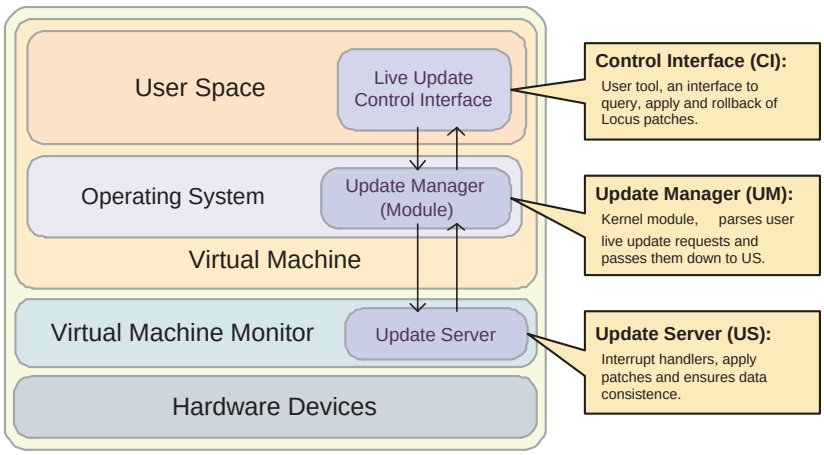

Figure 2. Architecture of LUCOS

while splitting a structure may need a one-to-multiple conversion. Being aware of these cases, LUCOS supports a state transformation between the old and the new data structures on a multiple-to-multiple basis. Therefore, for each data structure, a state transfer function should be provided to maintain the consistence between itself and other related data structures.

For some bug-fixing patches, simply replacing the buggy code may not solve the whole problem. For the example code in Section 3.2, some additional transfer function needs to be called to fix the deadlock condition. Therefore, LUCOS allows a patch to register some callback functions to be called after some specific situations. For example, after a thread leaves a function, and after all threads leave the patched functions.

Finally, a valid LUCOS patch file contains all or parts of the items listed below. It should be noted that the items marked with a star are optional. Hence, a broad range of kernel patches could be accepted, and patch developers will have more flexibility to tailor their patches.

- New *declarations* of *data structures*, including *single-instance* and *multiple-instance data structures* and the *functions* that manipulate them. These code and data will be used by Linux kernel after applying the patch.
- **Callback functions* that will be called after some specific events happen during the patch process. LUCOS currently supports three types of callbacks: (1) *function callbacks*, which will be invoked each time a thread leaves a function being patched; (2) *thread callbacks*, which will be called when all threads have left a function being patched; (3) *data callbacks*, which will be invoked when all threads using a data structure have left all the functions that manipulate the instance of the data structure.
- **Patch startup* and *patch cleanup functions*. Patch startup functions are responsible for some initialization needed to set up the environment for live updates, for example, initializing the added data structures and registering new resources to the kernel. Also, the patch startup functions will locate all instances of data structures if the patch updates data structures. On the other hand, patch cleanup functions carry out some cleanup work to complete the live update. Examples include freeing unused data structures and un-registering the old resources. To support rolling back a committed patch, the patch startup and cleanup functions should be designed with the ability to be reused for the rollback process. An argument to such a function indicates whether it is executed by a rollback process or not.
- **State transfer* functions that will be utilized to maintain the consistency of the old and the new data. Instead of a one-to-one correspondence, the state transfer functions could have a multiple-to-multiple correspondence. Therefore, writing to one instance of a data structures may trigger updates of multiple instances of multiple data structures.
- The *module_init* function and the *module_exit* function are similar to their counterpart in general Linux kernel modules. To prevent malicious or erroneous removal of a kernel module, a function call to the update manager is added to the module_exit function to prevent arbitrary removal of the patched module.

  To facilitate the construction of LUCOS patch files, we have provided a set of helper functions. These functions could be easily used by the patch developers to write initialization and finalization code, startup and cleanup functions, and insertion and deletion of callback and state transfer functions. Although it's generally impossible to completely automate the process of patch construction for a powerful dynamic update system, we plan to automate the tedious work by generating templates for the LUCOS patch files.

4.1.2 Generating LUCOS Patches

Figure 3 shows the work flow of constructing a LUCOS patch. After a patch file is ready, the standard module compilation proce-

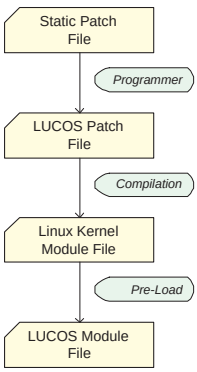

Figure 3. Work flow to construct a LUCOS patch

dure is initiated to generate the patch module. However, to prevent an arbitrary access to the kernel function and data, Linux exposes only a limited set of kernel data and functions to kernel modules.

Therefore, inserting a module that patches non-exposed functions and data will not be permitted by the kernel. LUCOS solves this problem by using a linker to pre-load the patch modules before inserting them into kernel. The linker scans the System.map file that contains all kernel symbols and relocates the non-exposed symbols in the patch module.

4.2 Performing Live Updates

Performing a live update involves three steps: (1) set up a patch; (2) apply the patch according to its type; (3) terminate the patch process. The following subsections detail the three steps.

4.2.1 Patch Setup

On receiving a live update request from the control interface, the update manager validates the request and gathers necessary information about the patch.

Binary rewriting is used to redirect a function call from the original function to the patched function. However, the variable-size instructions of the x86 architecture complicate the process of binary rewriting. According to x86's calling convention and register usage convention, the prologue of a function consists of instructions that save the *callee-saved* registers and adjust the stack size. Usually, each of these instructions is short and occupies no more than 5 bytes, e.g., the binary code of "*pushl %ebx*" is 0x53 and occupies only one byte. On the other hand, the length of a *jmp* instruction, which is used to redirect a call from the original function to the patched function, is 5 bytes. Therefore, it is necessary to make sure that no thread context or interrupt context is currently executing in the first 5 bytes of the function to be patched. This is accomplished by iterating all kernel threads of the operating system, and make sure that none of them is executing between the starting address of the function to be patched and 5 bytes beyond that address. Fortunately, this situation rarely occurs since the prologue instructions in a function are usually some short-term instructions (e.g. push, move and add), which is unlikely to be blocked.

Before applying patches to both code and data, we need to count the number of threads or interrupt contexts executing in the code to be patched. This is achieved by iterating the kernel stacks of all kernel threads. If no thread or interrupt context is executing in the code, then the patch process can be simplified. In this case, after invoking the state transfer function to transfer the state from the old data to the new data, the remaining live update procedure can be simplified to become a live update to code only. Otherwise, VMM is responsible for maintaining the coherence between the two versions of the data.

If a patch module provides the startup function, then it will be invoked in this phase and perform some initialization work to prepare for the patch. After all of the startup work is done, the update manager issues a hypercall to inform the update server to apply the patch.

4.2.2 Applying Patches

We do not rely on a quiescence state before a live update because hot spots in an operating system rarely enter a quiescent state. It will be very difficult to live update them if we have to wait for them to enter a quiescence state. As a result, LUCOS allows coexistence of both the old and the new versions of a data structure to be patched. Hence, some coherence mechanism is required to ensure their consistency. It seems that the easiest way to accomplish this is to write protect both versions of the data. When there is a write operation to either of them, it will be trapped to the update server where proper action could be taken for data synchronization. Traps and fine-grained protection mechanisms are required to implement efficient data synchronization.

Pentium provides two protection mechanisms known as segmentation and paging. Although the segmentation mechanism allows variable-length segments, it is difficult to change the segment selectors of the application programs on the fly. It makes a fine-grained data synchronization impractical. Page-based protection mechanism is thus used to implement the data synchronization.

The VMM will first write protect the original data and the newly introduced data when a live update begins. When a write access to either version of the data takes place, the access is trapped to the VMM. The VMM will then "unprotect" the page for the write access and set the single-step flag in x86. The write access to the page will become valid and could take place. A single-step debug exception returns control to the VMM after the write access, and the VMM invokes the state transfer function to ensure the consistency of the two versions of the data. When the debug exception returns, the operating system could resume its normal execution.

The details on how to live update the two types of patches are described as follows.

Live update to code only: This is the simplest case. Figure 4

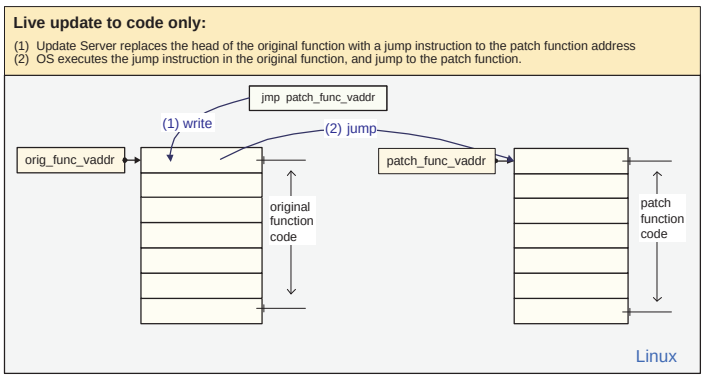

Figure 4. Live update to code only.

shows the steps to live update code only. On receiving an update request from the update manager, the update server replaces the first 5 bytes of the original function with a relative jump to the address of the new function.

Live update to code with data changes:

Figure 5 shows the steps in live updating reentrance code with data changes:

1. The update server replaces the prologue of the original function with a jmp instruction to the patched function. It then write protects the pages that contain the original and the new data.
2. Any update to the original or the new data will trigger a page fault to the update server.
3. The page fault handler notifies the update server if the fault is for a write protection, and the faulting address is within the protected page. The update server then restores the write-protected page to commit the effect of the faulting instruction. It also sets the single-step flag to allow the update server to regain control later. A single-step debug exception is triggered after the faulting instruction commits. The update server regains control and checks whether the faulting instruction accesses either version of the data. If so, the state transfer function is called to ensure the coherence between them. After that, the update server returns control to the operating system. It resumes the execution with the single-step flag cleared.
4. When the original data structure is no longer active, the update server restores the read/write flags of the pages that contain the original and the new data.

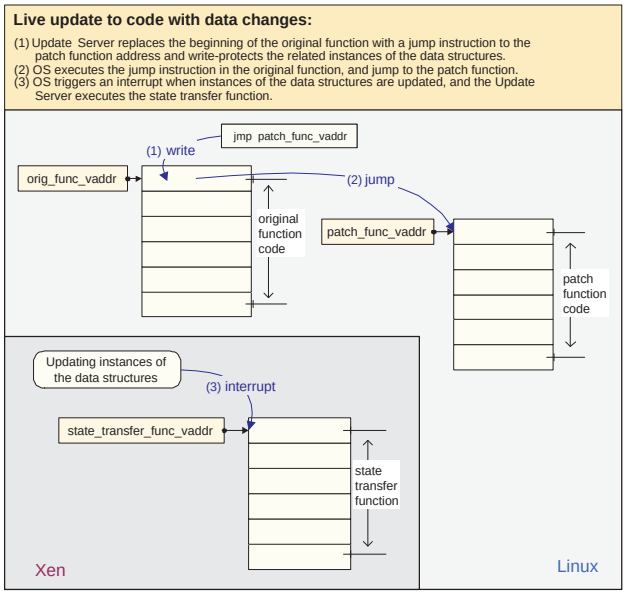

Figure 5. Live update to code with data changes.

Due to the fact that the CPU does not automatically maintain the coherence between the memory (including the cache) and the general purpose registers, updates to one memory location will not be visible to the corresponding loaded registers in time. In LUCOS, as the old and new versions data are logically viewed as one uniform entity by the kernel and protected from simultaneously accesses using some synchronization techniques(such as locks), such delay of loading will not happen. However, misuses of locks may prevent the old or new versions of data from receiving up-to-date values even if the state transfer function has been invoked. For instance, for a patch updating a global variable and the spinlock protecting the variable, the patch should provide a state transfer function to maintain the coherence between the two spinlocks, or the global variable will end up with inconsistency.

4.2.3 Patch Termination

Depending on the type of live updates, it needs different mechanisms to decide whether a patch process has completed or not. For live updates to code only, the patch process is completed and could be terminated as soon as the function redirection is done. However, for live updates to both code and data, deciding whether it is safe to terminate a live update process poses a special challenge. To safely terminate the patch process, we use a technique called *stack inspection*[15], to exam all kernel stacks and see whether there are threads still executing in the original functions.

In the patch setup stage, the update server iterates each kernel thread and determines if the thread executes within the function to be patched by inspecting the call trace of the thread stack, and adding it to a thread list if it does. Then, the update server replaces the return address of the original function with the address of a stub function. The stub function will remove the executing thread from the thread list, invoke the function callbacks, and return to the caller of the original function. On removing a thread from the thread list, the update server invokes the thread callbacks and checks whether the thread list becomes empty or not. If it is empty, the original function is no longer active. When all functions manipulating a data structure become inactive, the update server invokes the data callbacks and marks the data structure as unused. When all data structures being patched become inactive, the live update process is safe to be terminated. The update server will perform some cleanup work such as restoring the write-protected pages.

When the update server returns to the update manager with a patch-finished flag, the update manager knows that the whole update process has completed. It then performs some cleanup work and calls the patch cleanup function in the patch module to reclaim unused resources.

The stack inspection may suffer some scalability problems. However, in practice, it's quite unusual to apply live updates when the system is under extremely heavy workload.

4.3 Patch Rollback

The ability to rollback a committed patch gives system administrators significant flexibility to manage their operating systems. Although LUCOS assumes the correctness of the patches and leaves the verification to kernel developers and testers, LUCOS does support rolling back buggy patches.

From LUCOS's viewpoint, rollbacks are a special type of patches: using the original code and data to patch the committed ones. The original patch startup function and the cleanup function are reused with their arguments set to indicate their invocation in a rollback process. After restoring the first 5 bytes in the original function with their saved value, a function call to the patched function will be redirected back to the original function. If the patch to a rollback affects data, the corresponding state transfer function is reused to maintain the consistency during the patch process.

However, supporting rollbacks may incur some performance and resource overhead. For instance, the code of the original function needs to be kept in memory for possible rollbacks. This prevents some possible optimizations such as copying the code and data of a patched function to the space of the original function, which may improve its locality. The supplement functions such as patch startup functions, patch cleanup functions, state transfer functions and their related data structures, have to be kept in memory as well. For safety and security reasons, such overhead seems quite necessary. Perhaps, it is better to give administrators the flexibility of choosing the most suitable rollback strategy tailored to their needs.

4.4 Discussion and Further Work

4.4.1 State Synchronization

The shortcoming of the page-based protection mechanism in implementing data synchronization lies in its coarse granularity. Some novel architectures[16] support efficient fine-grained memory protection. They allow arbitrary permission control at the granularity of individual words. This feature greatly simplifies data synchronization. In an architecture that supports word-level memory protection, data synchronization will be more efficient and easier to implement.

4.4.2 Automated Patch Construction

Our current approach to generate dynamic patches is manually reading the static patches from kernel developers, deciding the data structures to be tainted, and write dynamic patch files from scratch. This approach requires some tedious engineering effort. For a powerful live update systems, it's generally impossible to completely automate the process of patch generation. Our future work will target at automating the majority of the work on transformation from static patches to LUCOS patches, leaving minimal work to the developers, such as providing callbacks and state transfer functions.

4.4.3 Virtualization Issues

Our framework does not compromise the OS transparency. However, because current version of Xen is a para-virtualized VMM,

the target operating system is a slightly modified version of Linux. Xen-3.0 is reported to support full system virtualization using a novel hardware feature called Vanderpool [17]. Our future work will include porting LUCOS to Xen-3.0.

4.4.4 Safety and Security in Live Update

It is essential that a live update should not taint or corrupt the running kernel. We currently assume that the patches are well tested and bug-free, and allow rolling back an existing patch if it is found to be buggy. However, if the kernel collapses when executing the buggy patched function, there will be no chance for a rollback.

One possible solution to this problem is to add an additional protection domain to Linux, similar to the approach in [18, 19]. A validation phase will be added to the live update process. The applied patch will be executed in the protection domain until it is proved to be correct. Even if the applied patch is malicious, the fault in the patch will not corrupt the entire kernel. If a patch is found to be buggy or malicious, the update manager could automatically rollback the patch and inform the administrator to fix the bug. When the applied patches are proved to be bug-free, the update manager could be informed to commit these patches by moving them from the protection domain to the kernel space. One downside of this approach is that kernel will suffer performance degradation during the validation phase. However, this tradeoff might be worthwhile since the kernel can survive a potential corruption.

5. Experimental Results

5.1 Experiences in Applying Live Updates

We used several typical patches selected from Linux kernel developers to measure the performance overhead imposed by LUCOS. Four of them were selected from patches that upgrade the Linux kernel from 2.6.10 to 2.6.11. As our implementation was specific to Xen, we also selected a patch from an upgrade of Xen-Linux.

1. Fixing the page reading bug: This is a simple case that only affects code, which fixes a page reading bug. As stated in the log of the patch[2], a concurrent read while invalidating pages could cause a read error because the invalidation could make the page out of date at the wrong time. It was solved by dynamically patching two functions which added an explicit check to see whether the page was invalidated. In this patch, as no global variable was involved, no state transfer was required. After the buggy functions have been replaced by the new functions, the bug was fixed.

2. Removal of livelock avoidance: As described in the patch file[3], It is generally believed that a seek after a read in a definite loop could result in a livelock for the *kjournald* kernel thread. The livelock avoidance code in *kjournald* may cause long latencies under some circumstances. However, with the dual list write-out arrangement in *kjournald*, this livelock will never occur. This problem was solved by removing the livelock avoidance code. As such a patch requires modifications to the code only, applying it is rather easy.

3. Upgrading the process scheduler: In Linux kernel 2.6.10, each task structure maintains a variable, named *interactive credit*. It gives an interactive task more priority bonus when it is scheduled. However, as it is stated in the patch logs[4], this mechanism did not consider tasks that have periods of being fully cpu-bound, and then put to sleep while waiting on pipes or signals. It could lead to a disproportionate share of cpu time for such tasks. We fixed it by patching the main schedule function and two other relevant ones. As no variable is added to the task structure, no state synchronization is needed. We simply discarded the *interactive credit*. However, at the patch startup time, all *interactive credits* in each task should be re-calculated to give them the priority bonuses.

4. Reconstruction of the IRQ descriptors: Interrupt service routines are organized as an array containing various information about the interrupts. It includes the interrupt status, the hardware they serve and all actions of each interrupt. As the hardware devices evolve, device driver developers require additional space to store private data for the interrupt handlers. The patch is to dynamically replace the original IRQ descriptor array with a new array that has an extra void pointer. All of the functions that manipulate the original array are replaced with the new patched functions. While there are threads executing in the original functions, data synchronization is performed between the original array and its new counterpart. After the last thread have left the original functions, all original functions are no longer active and data synchronization can be stopped. At that point, the whole live update process is finished.

5. Upgrading backend block device drivers in Xen-Linux: Xen hosts multiple operating systems concurrently. One privileged Xen-Linux can access a block device directly and provide services for other operating systems. *Blockback* is designed to support such a functionality. An array is provided for buffering all of the incoming disk requests that are waiting for the service of the dispatcher. In Xen-Linux 2.6.11, a new member is added to the structure of the pending requests to improve disk I/O performance. To apply such a patch, we first created a new array for the pending requests with the new structure in the patch. Then we replaced the function that manipulated the original array. After all threads have exited the function, the original array can be safely discarded.

5.2 Performance Evaluation

To measure the overall performance overhead of LUCOS, we compare Xen-Linux in LUCOS against native Linux and the original Xen-Linux, which is a variant of Linux ported to run on Xen VMM.

All the experiments were conducted on a system equipped with a 3.0GHz Pentium IV with 1GB RAM, a Intel Pro 100/1000 Ethernet NIC in a 100M LAN, and a single 250GB 7200 RPM SATA disk. The version of Linux and Xen-Linux is 2.6.10 and the version of Xen VMM is 2.0.5. The Fedora Core 2 distribution was used throughout. It is installed on ext3 file system. We configured 900,000KB of memory for each variant of Linux.

5.2.1 Relative Performance

A set of benchmarks were used to evaluate LUCOS's performance. As LUCOS is implemented on Xen, we tested four benchmarks that were also tested on Xen [8] as well: SPEC CPU 2000 [20] measures the performance of CPU-intensive workloads. Open Source Database Benchmark suite(OSDB) [21] tests the performance of PostgreSQL database, with the tests for both Information Retrieval (IR) and Online Transaction Processing(OLTP) workloads; Linux build measures the overall time to built a Linux Kernel 2.6.10 with gcc-3.3.3. For the experiment setup, we used OSDB-x0.15-1 in conjuction with PostgreSQL 7.3.6. All benchmarks were with their default configurations.

As depicted in Figure 6, the performance results between Xen-Linux and Xen-Linux of LUCOS are very similar, and LUCOS incurs a less than 1% performance lost. This reflects the fact that LUCOS is composed of a set of passive modules in Xen and Linux. Although there is some performance gap between LUCOS-Xen-Linux and Native Linux, we believe the overhead is acceptable to support live update features.

[2] http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.10/2.6.10-mm1/broken-out/readpage-vs-invalidate-fix.patch

[3] http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.10/2.6.10-mm1/broken-out/jbd-remove-livelock-avoidance.patch

[4] http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.10/2.6.10-mm1/broken-out/sched-remove_interactive_credit.patch

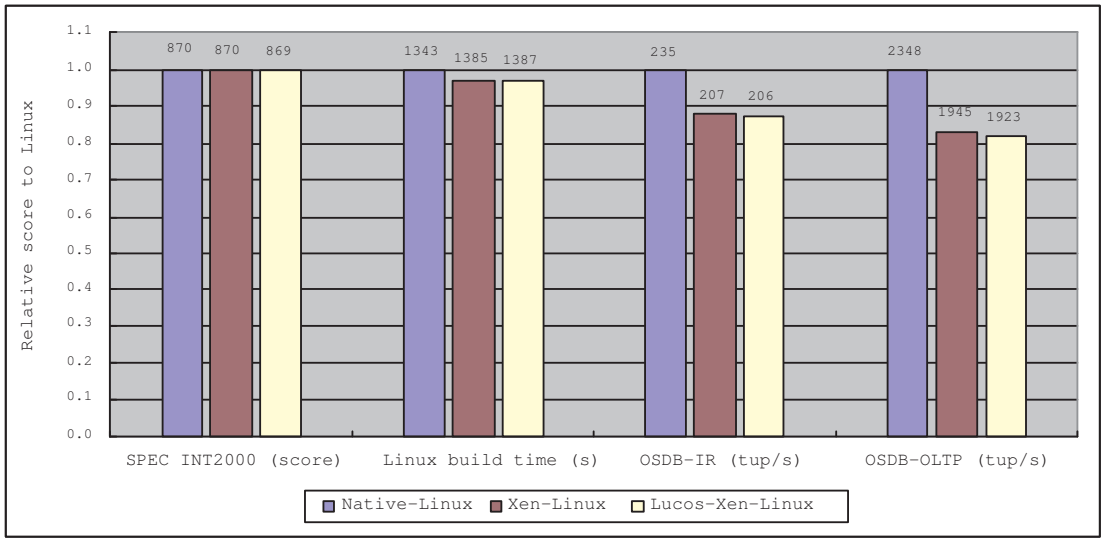

Figure 6. Relative Performance of Native Linux, Xen-Linux and LUCOS-Xen-Linux.

Table 1. Time to apply and rollback live updates.

No.	#Funcs	apply Time(in ns)	rollback time (in ns)
1	2	21,426	19,663
2	1	14,916	13,113
3	3	23,715	22,041
4	13	215,921	217,479
5	1	42,900	36,330

5.2.2 Time Consumed to Apply and Rollback Live Updates

The time to apply and rollback a patch was measured by reading the hardware cycle counter at both the beginning and the end of each live update. The time presented is the median of five trials. To better simulate the real-life patching scenario, the live update process were performed when the systems was running the OSDB online transaction processing simultaneously. Table 1 shows the corresponding time to apply and rollback patches mentioned in section 5.1. As depicted in the table, the time spent on applying a live update is relative short, even for live updates that require state synchronization between data.

For patches that only affect code, the time to do a live update is rather short as no state transfer is required in the live updating process. For patches that affect data structures, it tends to be more time-consuming. Actually, the time to apply such a patch is closely related to the time when all threads leave the original functions.

6. Related works

Our work differs from previous research effort in several aspects. LUCOS employs system virtualization to live update a running operating system. It eliminates the need for a quiescence state and allows live updating a running operating system on-demand.

K42 [3, 5, 4] is an object-oriented operating system. It provides live update capability by exploring techniques such as quiescence detection, state transfer, factory mechanisms, and state tracking. However, it is tightly bound to the object-oriented approach. Most non-object-oriented operating systems cannot benefit from these features. Furthermore, it requires that the components be in a quiescence state before they could be live updated.

Dynamic Kernel Modifier or DKM [22] allows modifying the execution of kernel functions in a user mode without the need of recompiling and modifying the kernel source. It is designed to enable rapid development and performance tuning for the Linux kernel. DKM supports many schemes to modify the kernel. They include inserting trace points, nullifying functions, and replacing functions. However, it does not support changes to the data structures. Therefore, live updating Linux kernel is not supported.

Dynamic software updating [23, 24, 25] provides application software with the ability to be upgraded without service disruption. However, these techniques cannot be easily applied to existing operating systems due to their complexity.

Linux kernel module [26] allows some specific parts of the kernel code and data (usually kernel module) to be updated on the fly, but with some strict constraints. Live updating is allowed only when the entire kernel is inactive, or when no other parts of the kernel threads are in the context of the code.

Read-Copy Update (RCU) [27] is a concurrent mechanism optimized for read/write locks. Readers can avoid acquiring any locks, while writers update their own private copies. All updates are committed in a quiescent state where all active operations have completed. Nevertheless, RCU has several limitations in providing live updating features to operating systems: it is a per data structure option; as the write operation is rather time consuming, it is only suitable for multiple-reads-few-writes cases.

Devirtualizable Virtual Machine [28] supports general, single-node, online maintenance by running enterprise applications that are serving requests on one virtual machine, while upgrading the OS, reconfiguring software, or updating applications on a second virtual machine simultaneously. Application migration tools are used to transfer state from the production VM to the upgraded VM, which will then replace the production VM. This approach is a good substitute for cluster-style maintenance. However, it requires two virtual machines to be active on the same machine simultaneously. It also needs special tools to migrate applications from one to another.

7. Conclusion

We propose using virtualization to live update a running operating system on demand, without the requirement of a quiescence state.

The prototype we have implemented, named LUCOS, is able to live update the Linux without disrupting its services and with minimal overhead during the normal execution. We demonstrate this approach by applying several real-life Linux kernel patches on the fly. Performance measurements showed that our implementation incurs negligible performance overhead compared to a Xen-Linux.

Acknowledgments

This work and this paper would not have been done without the efforts of others. We would like to specially acknowledge our teammates Pengcheng Liu, Jie Yu, Hai Du and Tong Sun for their hard work on building the prototype system and doing the experiments. We also thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] David A. Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, March 2002.

[2] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software.*, 10(2):53–65, 1993.

[3] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, pages 141–154. USENIX Association, June 2003.

[4] Craig A. N. Soules Robert W. Wisniewski Dilma Da Silva Orran Krieger Marc Auslander David Edelsohn Ben Gamsa Gregory R. Ganger Paul McKenney Michal Ostrowski Bryan Rosenburg Michael Stumm Jimi Xenidis Jonathan Appavoo, Kevin Hui. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, January 2003.

[5] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Jeremy Kerr, Orran Krieger, and Robert W. Wisniewski. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–291, Anaheim, CA, USA, June 2005. USENIX Association.

[6] Robert Wisniewski Marc Auslander David Edelsohn Ben Gamsa Orran Krieger Bryan Rosenburg Kevin Hui, Jonathan Appavoo and Michael Stumm. Supporting hot-swappable components for system software. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII 01)*, pages 170–170, Elmau/Oberbayern, Germany, May 2001.

[7] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[8] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, October 2003.

[9] Robert Rose. Survey of system virtualization techniques. http://citeseer.ist.psu.edu/720518.html, March 2004.

[10] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th symposium on Operating systems design and implementation,OSDI'02*, pages 181–194. USENIX Association, December 2002.

[11] Microsoft Corporation. Microsoft virtual server 2005. http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx.

[12] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, pages 195–209, Boston, MA, USA, October 2002. USENIX Association.

[13] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Monterey, CA, USA, June 2002.

[14] Jeff Dike. User-mode linux. In *Proceedings of the 5th Annual Linux Showcase & Conference*, Oakland, CA, November 2001. Usenix Association.

[15] Paul Burstein Gautam Altekar, Ilya Bagrak and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of 14th USENIX Security Symposium*, Baltimore, MD USA, 2005.

[16] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, volume 37, 10 of *ACM SIGPLAN notices*, pages 304–316. ACM Press, 2002.

[17] Intel Cooperation. Intel vanderpool technology for IA-32 processors (VT-x) preliminary specification. http://www.intel.com/technology/computing/vptech/.

[18] Brian N. Bershad Michael M. Swift and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222, Bolton Landing, NY, USA, October 2003. ACM Press.

[19] Brian N. Bershad Michael M. Swift, Muthukaruppan Annamalai and Henry M. Levy. Recovering Device Drivers. In *Preceedings of 6th Symposium on Operating Systems Design and Implementation*, pages 1–16, San Francisco, CA, USA, 2005.

[20] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[21] Maciej Suchomski Peter Eisentraut Justin Clift Andy Riebs, Mark Kirkwood and Paul Wagner. Osdb x0.15-1. http://osdb.sourceforge.net.

[22] Ronald G. Minnich. A dynamic kernel modifier for linux. In *Proceedings of the LACSI Symposium*, September 2002.

[23] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[24] Robert Bialek and Eric Jul. A framework for evolutionary, dynamically updatable, component-based systems. In *24th International Conference on Distributed Computing Systems Workshops - W2: DARES(ICDCSW'04)*, pages 326–331, 2004.

[25] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proceedings of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Long Beach)*, pages 183–194, January 2005.

[26] Matt Welsh. Implementing loadable kernel modules for Linux. *j-DDJ*, 20(5):18–20, 22, 24, 96, April 1995.

[27] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[28] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review*, 38(5):211–223, December 2004.