Seamless Kernel Updates

Maxim Siniavine, Ashvin Goel
University of Toronto

***Abstract*—Kernel patches are released frequently to fix bugs and security vulnerabilities. However, users and system administrators often delay installing these updates because they require a system reboot, which results in disruption of service and the loss of application state. Unfortunately, the longer a system remains out-of-date, the higher is the likelihood of system failure or a successful attack. Approaches, such as dynamic patching and hot swapping, have been proposed for updating the kernel. All of them either limit the types of updates that are supported, or require significant programming effort to manage.**

We have designed a system that checkpoints application-visible state, updates the kernel, and restores the application state thus minimizing disruption of service. By checkpointing high-level state, our system no longer depends on the precise implementation of a patch and can apply all backward compatible patches. Our results show that updates to major releases of the Linux kernel can be applied with minimal effort and no observable overhead.

I. Introduction

Operating system maintainers release kernel patches regularly to fix security vulnerabilities and bugs, and to add features. However, users and system administrators often delay installing these updates because they require a system reboot, which results in disruption of service and the loss of application state. For example, updating the operating system for a game server typically requires scheduled server downtimes, during which time all users stop playing the game, wait for the server to come back up, login to the server, and then play the game from the beginning, which is especially annoying for shooter and other real-time games. Unfortunately, the longer an out-of-date system remains operational, the higher is the risk of a bug being triggered, or a system being exploited, since most exploits target existing vulnerabilities. In addition, users are unable to use the new features, e.g., performance optimizations, available in the kernel updates.

Realizing these problems, application programmers are increasingly designing programs that can be updated without significant disruption. For example, users of web applications are not aware when the application is updated and can start using the new version after simply reloading the page. In fact, typically users have no control over updates, which helps avoid the need to support several application versions. Similarly, many large applications save and restore their state on an update (e.g., browsers restore web page tabs), thereby reducing disruption. Today, operating system kernels are a major component of the software stack that require significant time for updates and lose state after an update.

Existing kernel update systems work at varying granularity. Dynamic patching performs updates at function granularity [1], [2], and hot swapping at object or module granularity [3], [4]. These techniques require significant programmer effort for implementing patch, object or module-specific state transfer functions that synchronize the state of an updated component with an existing component. For example, hot patching operates at function granularity and can be applied relatively easily to patches that only change code. However, carefully crafted state transfer functions are needed for patching updated data structures. Similarly, object and module granularity update systems require component-specific transfer functions for the updated stateful components, and must be designed to handle changes to the component interfaces [4].

None of these techniques handle cross-cutting changes due to major restructuring of code that occurs across major kernel revisions. For example, the Linux kernel is updated with five patches every hour on average, and developers release a major kernel release every 2-3 months [5]. Later in the paper, we show that these releases often consist of over a million lines of modified or new code. Requiring programmers to write and test state transfer functions for their patches is simply infeasible, especially when kernel patches occur so frequently and major revisions involve millions of lines of code.

Our goal is to install major kernel updates reliably, with minimal programmer effort, and without requiring user intervention or any changes to applications. The main insight of this work is that applying updates at a courser granularity reduces programming effort. In particular, updates performed at a higher level of abstraction hide implementation details, reducing the need to write state transfer functions for each patch. For example, say that a transfer function exists for a stateful module. A patch that changes module internal state will not require an *additional* transfer function because this state is not exposed, making the patch easier to apply. Similarly, Swift et al. update device drivers at multi-module granularity by using common driver interfaces to automatically capture and transfer state between driver versions [6], [7].

Taking this idea to the limit, we have designed a kernel update system for Linux that checkpoints application-visible state, reboots and updates the entire kernel, and restores the application state. The checkpointed state consists of information exposed by the kernel to applications via system calls, such as memory layout and open files, and via the network, such as network protocol state. Our update system requires the least amount of additional programmer effort for installing a patch, because it hides most kernel implementation details, including interfaces between the kernel components. Furthermore, the kernel and the applications are strongly isolated from each other by memory management hardware and communicate by passing messages, i.e., system calls. As a result, there is no need to detect and update references from old to new data structures, or determine when the update process can terminate, all of which pose significant challenges in dynamic patching systems. Another significant benefit is that our system can handle all patches that are backward compatible at the system call and network protocol level, because they do not affect application-visible state. Kernel patches generally provide such compatibility to minimize disruption. The main drawback of rebooting the kernel is that it is human perceptible, but we believe that the main impediment to applying updates today is loss of application state, rather than brief system unavailability.

Our focus on designing a reliable and practical update system raises several challenges. Ensuring that the system will restore applications reliably requires taking a consistent checkpoint. When kernel data structures are inconsistent, e.g., when a system call is in progress, a consistent checkpoint cannot be taken. Waiting for system calls to finish is unreasonable since many system calls can block in the kernel indefinitely. Another option is to interrupt system calls, but many applications are not designed to handle interrupted calls. Instead, we start with the POSIX specification for restarting system calls when a signal occurs, and provide a method for resuming system calls transparent to applications. Unlike dynamic patching and hot swapping methods, our solution guarantees quiescence, allowing consistent checkpoints to be taken for all updates.

The second challenge is that a practical system should require minimal programmer effort for applying kernel updates. To achieve this goal, the checkpoint format and the checkpoint/restore procedures must be made as independent of the kernel implementation as possible. We checkpoint data in the same format as exposed by the system call API and the network protocols. Both are standardized, and so our checkpoint format is independent of the kernel version, and we expect it to evolve slowly over time. An additional benefit of this approach is that we can use existing kernel functionality to convert the data to and from the kernel to the checkpoint, since this functionality is already needed to perform these conversions during system calls. When the kernel is updated, the updated functions will perform the conversion correctly. To minimize changes to the checkpoint procedures, we use kernel API functions as far as possible. These include system call functions and functions exported to kernel modules, both of which evolve slower than internal kernel functions.

This work makes three contributions. First, we design a reliable and practical kernel update system that allows taking a consistent checkpoint for all kernel updates, and requires minimal programmer effort for applying these updates. Second, we perform a detailed analysis of the effort needed to support updates across major kernel releases, representing more than a year and a half of changes to the kernel. During this time, six million lines of code were changed in 23,000 kernel files. We are not aware of any system that provides such extensive support for kernel updates. Finally, we evaluate our implementation and show that it works seamlessly for several, large, real-world applications, with no perceivable performance overhead, and without requiring any application modifications. The overall reboot time is reduced by a factor of 4 to 10 for several of these applications.

Section II presents our approach. Section III describes the implementation of our system. Section IV presents our analysis of the programmer effort needed to use our system and evaluates the performance of the system. Section V discusses related work in this area. Section VI summarizes our work and provides conclusions.

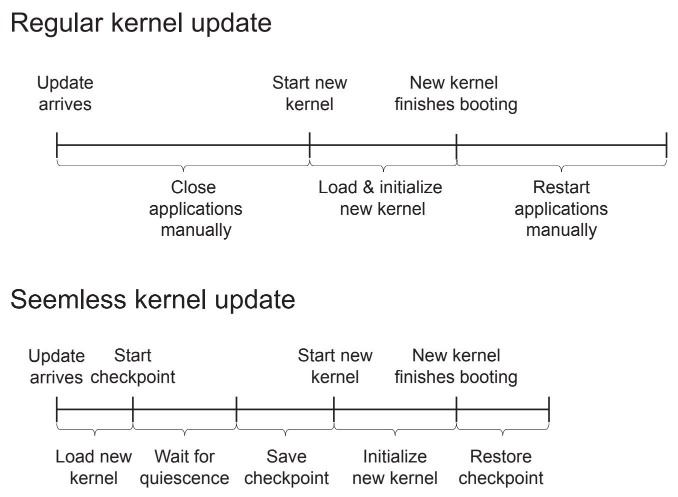


Figure 1. Time line for regular and seamless kernel update

II. Approach

Our goal is to perform kernel updates seamlessly, without requiring user intervention or any changes to applications. Figure 1 shows the time line of events for regular updates and seamless updates. During a regular update, applications are closed manually after saving work, the kernel is rebooted, and then applications need to be restarted manually. A seamless update treats the kernel as a replaceable component, allowing updates to the kernel without affecting application state. It operates in five steps as shown in Figure 1.

1) **Load new kernel:** When a kernel update arrives, we use the kexec facility in Linux to load and start executing the new kernel image. We modified kexec so that it performs the next two steps before starting the new kernel.
2) **Wait for quiescence:** We ensure that the kernel reaches quiescence, as described in Section II-A.
3) **Save checkpoint:** The checkpoint code walks the kernel data structures associated with application-visible state and converts them to a high-level format that is independent of the kernel version, as described in Section II-C.
4) **Initialize new kernel:** The kexec call jumps execution to the beginning of the new kernel, and the new kernel initializes itself.
5) **Restore checkpoint:** After kexec has initialized the new kernel, it reads the checkpoint and recreates applications using the checkpoint information. Then it restarts these applications, which may require restarting blocked system calls as described in Section II-B.

The three main challenges with seamless updates are achieving quiescence, restarting system calls and using a checkpoint format that is minimally dependent on the kernel implementation, as described below.

A. Quiescence

Ensuring that our system will restore applications reliably requires taking a consistent checkpoint, which imposes two conditions: 1) all threads are stopped, and 2) the kernel data structures are consistent. The first ensures that thread state remains consistent with its checkpoint state. For example, if a thread continues execution during or after checkpointing, it can affect the state of other threads with which it shares any resources. When both these conditions are met, we say that the kernel is quiescent, and a checkpoint can be taken.

The first condition can be easily met by pausing all processors other than the one running the checkpoint thread.[1] For the second condition, data structures can be inconsistent when any kernel code is executing, including in system calls, exception handlers and interrupt handlers. We need to let all kernel code finish executing and stop further entry into the kernel. However, system calls and exception handlers can block or sleep while waiting for events. The Linux kernel allows threads to sleep in one of two states, uninterruptible and interruptible sleep. During an uninterruptible sleep, the thread can hold locks and modify data structures, and so we allow this code to finish execution. Fortunately, an uninterruptible sleep is used for relatively short operations, such as a disk access for paging and memory allocation. Waiting for this code to finish execution has minimal impact on the overall update time because it is much faster than the time needed to initialize the new kernel.

A thread in an interruptible sleep can block indefinitely, e.g., waiting for user or network input. We cannot wait for these threads to continue before applying a kernel update. Fortunately, in this case, the kernel releases locks so that the thread does not block other threads from making progress, and ensures that its data structures are consistent before putting a thread in an interruptible sleep state. Furthermore, only system call threads can sleep in an interruptible state, and they can be interrupted by sending a signal to the thread, in which case, the system call returns immediately with an EINTR error code. To avoid blocking an update indefinitely, we save the state of system calls blocked in an interruptible sleep (below, we will call these blocked calls) and handle them during restore, as described in Section II-B.

We use the following steps to ensure that the kernel is quiescent before taking a checkpoint:

1) **Stop other processors:** We start the process by using the stop_machine function in Linux that allows waiting for interrupt handlers to finish executing on all processors, disables interrupts and pauses execution all other processors, and then returns control back to the calling processor where this code can continue running.
2) **Quiesce user threads:** In this step, we wait until *all* user threads are quiesced. If a user thread is currently in user mode or is blocked, then it is not running any kernel code, and we say that it is quiesced. When all user threads are quiesced, we return from stop_machine and proceed to Step 3, otherwise we perform the following steps:

   a) **Disable all further system calls:** We need to let threads running in the kernel or sleeping in uninterruptible sleep continue execution. By disabling further system calls, we can guarantee quiescence of user threads. If a thread issues a system call, we block it in a special blocked state, so that we know that it simply needs to be restarted on restore.

   b) **Wait briefly for quiescence:** We enable the processors again by returning from stop_machine, wait briefly for 20 ms, and then restart the process by returning to step 1. During the 20 ms wait, the kernel operates normally, allowing interrupts to occur and threads to exit the kernel or to block in interruptible sleep.
3) **Quiesce kernel threads:** In this step, we wait until *all* kernel threads are quiesced. Kernel threads are used to perform deferred processing tasks, such as writing dirty file buffers to disk. We separate quiescing user threads from quiescing kernel threads so that any existing deferred tasks can be completed before taking a checkpoint. Quiescing kernel threads proceeds in four steps:

   a) **Deferred processing:** At this point, we have returned from the stop_machine function and all interrupts are enabled, and we can perform various deferred processing tasks. For example, we create hard links to temporary files so that they are not removed after a kernel update and then save all dirty file buffers by calling the system-wide sync operation to complete any buffered IO.

   b) **Reboot notification:** Then, we send a standard reboot notification to all kernel threads so that these threads can prepare for devices to be shutdown. In particular, after the notification returns, the threads do not access any devices. For example, fast devices, such as the hard drive, will not serve any further requests because users threads are quiescent and the kernel threads have been notified about the reboot in this step.

   c) **Shutdown devices:** Next, we shutdown all devices because many drivers assume upon start-up that devices have previously been reset [8]. This shutdown process may wake up certain kernel threads (e.g., when the hard drive cache is flushed) and hence we cannot disable interrupts or suspend kernel threads until this point. However, since interrupts are enabled, certain slow devices, such as the keyboard, mouse, timer and the network may generate exogenous interrupts until the devices have been shutdown, but we prevent these interrupts from waking up sleeping user threads. Losing these exogenous interrupts will resemble a short system freeze, but without significantly affecting applications. For example, TCP packets will be retransmitted since we restore TCP state.

   d) **Wait briefly for quiescence:** After all the user threads are quiescent and all the devices have been shutdown, the kernel threads should not have any further work and they should not be running, i.e., they should be quiescent. At this point, we invoke stop_machine to disable interrupts again. For safety, we check if all the kernel threads are blocked, and if so, we can proceed to the next step. Otherwise, we return from stop_machine, wait briefly for 20 ms, and repeat Step 3.
4) **Take checkpoint:** Now that the kernel is quiescent and all interrupts are disabled, we can checkpoint application-

[1]The quiescence conditions don't apply to the checkpoint thread because it is not checkpointed and it doesn't modify any kernel data structures.

visible state. We ensure that the checkpoint accesses memory but not the disk, which has been shutdown in the previous step. The checkpoint and restore process is described in more detail in Section III.

B. Restarting System Calls

After restoring the checkpoint in the updated kernel, we need to resume thread execution, which requires handling system calls that were blocked. There are over 300 system calls in Linux, but only 57 of them are interruptible. [2] We do not need to consider uninterruptible systems calls because we had waited for them to finish executing before reaching quiescence. This is fortunate because many of these calls modify kernel data structures in complicated ways that are not idempotent and thus are not easily restartable.

For the interruptible calls, a simple solution would be to return the EINTR error code, since this return value is part of the specification of what happens when a signal is sent to the thread. However, most applications do not handle interrupted system calls, specially if they don't use signals.

Instead, we reissue the system calls that were blocked after the application is restored. To ensure correct behavior, we looked at the POSIX specification for restarting system calls upon a signal, since this specification is implemented by the Linux kernel. The kernel can already automatically restart some system calls when they are interrupted by a signal. It can restart these calls because their behavior is idempotent when they are blocked (even if they are not idempotent otherwise). In particular, the system calls block in order to wait for external events. However, if the event has not occurred, then the system call has not done any work, and so it can be safely reissued. The POSIX specification disallows restarting some of the system calls on a signal, for two reasons: 1) a timeout is associated with the system call, or 2) the system call is used to wait for signals. In our case, we still wish to restart these calls to avoid failing the application. For timeout related calls, they can be reissued after taking the timeout period into account, as discussed below. For signal related calls, they can be restarted, because a signal was never delivered in our system. However, they require adjusting the signal mask, as described below. We use five methods to transparently resume all the blocked system calls after restoring a user thread. A few system calls require multiple methods described below.

1) **Restart:** The system calls that are idempotent when they are blocked are restarted. This is exactly the same behavior the kernel already implements for these system calls when they are interrupted by a signal. Examples are open, wait and its variants, futex, socket calls such as accept, connect, etc. There are 19 such restartable calls.
2) **Track progress:** System calls that perform IO operations like read and write keep track of how much progress they have made. When these calls are interrupted by a signal, their current progress is returned to the user. We implement the same behavior, and after restoring the thread, we return the progress that the system call had made before the checkpoint was taken. We do not reissue this call to complete the operation (e.g., finish a partial read) because input may never arrive. It may appear that returning a partially-completed IO operation may cause certain applications to malfunction. However, system call semantics require correctly designed applications to handle short reads and writes. For example, on a read, fewer bytes may be available than requested because the read is close to end-of-file, or because the read is from a pipe or a terminal. Similarly, network applications communicate with messages of predetermined length and reissue the calls until the full message is processed [9]. As a result, we have not observed any problems with returning partial results for reads and writes for the applications that we have tested. When no progress has occurred, we restart the system call, because returning a zero indicates that the communication has terminated (EOF) after which the application may fail, when in fact our checkpoint maintains the communication channel. In this case, we can still safely reissue the call since it had not made any progress before being blocked. There are 23 calls that require progress tracking.
3) **Return success:** System calls that close file descriptors like close or dup2 invalidate the descriptors if they block. For these calls, we return success because when the checkpoint is taken, the file descriptor is already invalidated and the resource will be reset once the kernel is restarted. There are 3 such calls.
4) **Update timeout:** If the system call has a timeout associated with it, e.g., select, and it uses a short timeout compared to the time it takes to restart the kernel, we simply reissue the system call to avoid returning a spurious timeout. For long timeouts, we restart the system call after calculating the remaining time and subtracting the total time it took for the kernel to reboot and restore the thread. There are 11 calls that require timeout handling.
5) **Undo modifications:** Certain system calls, like pselect and ppoll, make a copy of the process signal mask, and then temporarily modify it. Before restarting these system calls, the signal mask has to be restored from the copy to the original state. The pselect and ppoll calls also require timeout handling. There are 7 calls that require undo modifications.

C. Checkpoint Format and Code

We checkpoint application-visible state, consisting of information exposed by the kernel to applications via system calls, such as memory layout and open files, and via the network, such as protocol state for network protocols implemented in the kernel. Checkpointing this state requires programmer effort proportional to the system call API rather than the size of the kernel implementation or the number of kernel updates. Furthermore, since the system call API and the network protocols are standardized and change relatively slowly over time, we expect that a carefully designed checkpoint format will evolve slowly.

[2] We found this number by manual analysis, and by cross correlating with the manual pages of the calls. Of the 57, several are variants of each other, such as the 32 and 64 bit versions of the call.

Structure Fields	Notes
In checkpoint	
vm_begin, vm_end	Region of address space controlled by this vm_area_struct
vm_page_prot	Address space is writable, readable or executable
vm_flags	Special attributes: for example direction the stack grows
vm_file	Name of the file mapped by a vm_area_struct
vm_pgoff	Offset from the beginning of the file
vm_private	Used for mmap operations
anon_vm	Specifies the type of reverse mapping used
Not in checkpoint	
mm_struct	Pointer to memory descriptor
vm_next	Pointer to the next vm_area of the process
vm_rb	Tree node used to find vm_area based on virtual address
vm_ops	Pointer to functions operating on vm_area_struct
vm_set, prio_tree_node	Used to implement reverse mapping
anon_vma_node, anon_vm	Used to implement reverse mapping

Table I
ANALYSIS OF VM_AREA_STRUCT

Our approach raises several issues: 1) what state should be saved, 2) the format in which it should be saved, 3) how sharing relationships between threads and their resources are expressed, and 4) how the code should be implemented. We save information available to the user space through system calls and via special file systems like /proc and /sysfs. We also save network protocol state, including buffered data, to ensure that a kernel update is transparent to network peers. For example, we store port numbers, sequence numbers and the contents of the retransmit queue for the TCP protocol.

The Linux kernel stores all process related information in the task_struct data structure. This structure contains process information such as the PID of the process, the parent and the children of the process, scheduling parameters and accounting information. The task structure contains pointers to other data structures that describe the resources currently being used by the process, such as memory management information, open files, etc. Thus the state of a process can be checkpointed by traversing the graph rooted at the task structure associated with the process. Our checkpoint generally saves the fields in the data structures that are visible to applications through system calls. These fields also allow us to restore these data structures during the restore process.

The checkpoint consists of a list of entries, representing either a thread or a resource owned by the thread, such as open files and sockets, with each resource using a unique format. As an example, Table I shows all the fields in the vm_area_struct structure in the kernel and the fields that are saved in our checkpoint. This structure represents a region of an application's address space, and the fields saved in our checkpoint are exposed to applications via the smaps file in the /proc file system or when accessing memory. For example, this information determines whether a memory access will cause an exception or a memory mapped file to be read from disk. The data in the checkpoint allows recreating the virtual memory region correctly, while the rest of the fields relate to the data structures used to implement the regions. The implementation dependent fields are not visible to the user, and thus not included in the checkpoint. We expect that while these fields may change (and have changed) over time, the checkpoint fields are unlikely to change significantly for backward compatibility.

Since we are saving state visible at the system call API, we save it in the same format. Internally, the kernel may store this state in any implementation-dependent way, but it needs to convert it when communicating with user applications. For example, a file path is a string in user space, but the kernel represents it by a sequence of dentry, qstr and mnt_point structures. By using a string for a file, we expect that the checkpoint will not depend on the kernel version, and we can use existing kernel functions to convert to the correct implementation-dependent kernel versions of the file-related structures. For example, the Linux do_filp_open function converts a path name to a file descriptor, and since it is the same function used to implement the open system call, we expect it to perform any implementation-dependent work required when opening a file.

Beside a portable checkpoint format, the checkpoint code must also be easy to port across different kernel versions for our update system to be practical. Ideally, the checkpoint mechanism would be implemented entirely in user space, relying only on the stable system call API. Unfortunately, some of the required functionality, such as page table information, and resource sharing relationships are only available in the kernel. Our code mostly uses functions exported to kernel modules, which evolve slower than internal kernel functions. We use the highest level functions available in the kernel as possible for saving and restoring state. For example to restore a pipe between two processes we call a high-level function do_pipe_flags which performs all the implementation dependent work needed to create a pipe. Afterwords, we use another high-level function to assign the newly created pipe to the two processes we are restoring. The high-level API takes care of all the details involved with maintaining the file descriptor tables of the two processes. More importantly, updates to the implementation of these functions will not affect our code. In essence, we take advantage of existing kernel functionality to implement state-transfer functions for kernel updates. We also do not rely on any virtualization or any indirection mechanism [10], [11], which would itself need to be maintained across kernel updates. Section IV-A analyzes our checkpoint format and code in more detail.

III. IMPLEMENTATION

This section describes the implementation of our kernel update system. We have added two system calls for executing the update process. They enable checkpointing specific processes and their children, and restoring all checkpointed processes. We have also added two debugging system calls that 1) determine whether a checkpoint is available, and 2) help distinguish between a process that was started normally or was restored from a checkpoint. All these system calls are intended to be used by scripts for managing the kernel update process. We do not require changes to existing programs.

The checkpoint operation involves saving data structure values and is relatively simple. When multiple processes share a resource, e.g. a memory region, they keep pointers to the same structure, e.g., a memory region descriptor. We implement this sharing by saving each resource separately in the checkpoint, and using pointers within the checkpoint to indicate the sharing. Each resource is tracked in a Save hash table. The key of this hash table is the memory address of the resource, and the value is the memory address of the corresponding entry in the checkpoint. When checkpointing any resource, we check if its address exists in the Save hash table, and if so, we use the value in the Save hash table to create a pointer to the existing checkpoint entry.

The restore operation is more complicated because it requires recreating custom processes from the checkpoint, similar to bootstrapping the initial user process in a kernel. The checkpoint information and the memory pages of all the processes need to be preserved during the reboot process. When the Linux kernel starts executing, it uses a bootstrap memory manager to dynamically allocate memory that is needed during the boot process before the memory management system has been initialized. After the bootstrap memory manager is initialized, we read the checkpoint and mark all the pages used by the checkpoint and valid process pages as reserved so that these pages cannot be immediately reused. After the restore operation, the process pages are marked as allocated and can be freed when a process terminates or as a result of paging.

During restore, we create a Restore hash table with keys that are the values from the Save hash table (i.e., the addresses of the checkpoint entry values). As each resource is restored, its memory address is filled in as the value in the Restore hash table. Looking up the Restore hash table as each resource is created ensures that the resource sharing relationships are setup correctly. The restore process runs with root privileges and hence care must be exercised when restoring the state of the OS resources. The kernel uses two types of credentials, one set for processes and another for files. We set the the various user and group IDs for each restored process thus ensuring that the restored process runs with the correct credentials. The restore process does not create files and hence we do not need to set up or modify any file credentials.

Our primary aim is to support as many commonly used server-side applications as possible. Our implementation currently checkpoints thread state, memory state, open files, network sockets, pipes, Unix sockets, SYSV IPC and terminals. It also checkpoints the state of typical server-side hardware such as a frame buffer, mouse, and keyboard. We have tested our system by updating the kernel while running several server applications, as described in Section IV. Our system aims to ensure that the client or the server will not notice the restart of the other. For TCP connections, any delays are handled by TCP retransmissions. For UDP connections, consider a restart of a game server. The client may miss updates from the server and get out-of-sync, but it will discard its local state after it receives an update from the upgraded server. We essentially take advantage of the typical fault tolerance mechanisms employed by client-server applications for handling unreliable connections or servers.

We have also tested our system for desktop machines by using the simple Xfbdev X server, the Twm window manager and several X programs. The mouse, keyboard, console and the graphics work correctly after the update, without requiring any application modifications or user intervention. Interestingly, the mouse and keyboard were initially freezing after an update if we kept moving the mouse or typing on the keyboard while the update occurred. This problem was solved when our quiescence algorithm was implemented correctly, showing its importance for kernel updates.

Below, we describe the checkpoint and restore implementation for thread, memory and file state. Then, we discuss some of the limitations of our current implementation. The other resources, especially TCP sockets, present several challenges for checkpointing, and are described in detail in an extended version of this work [12].

A. Threads

After quiescence is reached, the update thread stores the context switch data, in particular the register values and segment descriptor table entries for each thread, in the checkpoint. To restore a thread, we spawn a kernel thread for each thread stored in the checkpoint. Within the context of each spawned thread, we invoke a function that is similar to the execve system call. The execve system call replaces the state of the calling process with a new process whose state is obtained by reading an executable file from disk. Our function converts the kernel thread into a user thread by loading the state from the checkpoint in memory. We restore the saved register values and segment descriptors for the thread so that the updated kernel's context switch code can use these values to resume thread execution.

We restore the saved task_struct fields and reestablish the parent-child process hierarchy by changing the parent and real_parent pointers so that they point to their restored parents. We also make sure that all restored children are added to the list of children associated with their parent, and for multi-threaded processes we add threads to their thread group lists. After this setup, the kernel starts identifying the spawned kernel thread as a regular user process.

B. Address Space

An address space consists of a set of memory mapping regions and the page tables associated with one or more threads. Each memory mapping region describes a region of virtual addresses and stores information about the mapping such as protection bits and the backing store. The page table stores the mapping from virtual pages to physical pages. Currently, our implementation supports the x86 architecture, in which the page table structure is specified by the hardware and thus will not change across kernel versions.

Linux manages memory mapping regions using a top-level memory descriptor data structure (mm_struct) and one or more memory region descriptors (vm_area_struct). We store various fields associated with these data structures, including the start and end addresses of each memory region, protection flags, whether or not the region is anonymous, and the backing store, as shown in Figure I. For memory mapped files, we store the file name and the offset of the file for the virtual memory region. We restore these data structures by using the same functions that the kernel uses for allocating them during the execve (mm_struct), mmap (vm_area_struct) and mprotect system calls. These functions allow us to handle both anonymous regions and memory-mapped files. For example, we restore a memory-mapped file region by reopening the backing file and mapping it to the address associated with the region. The memory region structures can be shared and we handle any such sharing as described earlier.

The x86 architecture uses multi-level pages tables, and the top-level page table is called the page table directory. This page table format will not change across kernel versions and so we do not copy page tables or user pages during the checkpoint and restore. As a result, a process accesses the same page tables and physical pages before and after the kernel update. However, one complication with restoring page tables is that the Linux kernel executes in the address space context of the current user thread, and it is mapped at the top of the virtual address space of all processes. The corresponding page table entries for each process need to be updated after the kernel update. These page table entries are located in the page table directory. The function that creates the memory descriptor data structure (mm_struct) also initializes the page table directory with the appropriate kernel page table entries. We initialize the rest of this new page table directory from its pre-reboot version and then discard the pre-reboot version. At this point, we notify the memory manager to switch all process pages from being reserved to allocated to the new process.

C. Files

The Linux kernel uses three main data structures to track the files being used by a process. The top-level fs_struct structure keeps track of the current working directory and the root directory of a process, which can be changed with the chroot system call. The file descriptor table contains the list of allocated file descriptors and the corresponding open files. Finally, the file descriptor structure stores information about each open file. All three structures can be independently shared between several processes. For example, two processes might share the same working directory, may have different file descriptor tables, and yet share one or more opened files.

For each process, we store its root and current working directory, list of open file descriptors, and information about open files. Linux stores the current root and current working directory of a process as dentry structures. In the checkpoint, we store them as full path names. For files, we store its full path, inode number, current file position, access flags, and file type. Each file structure has a pointer to a dentry structure, which stores the file name. Each dentry structure has a pointer to another dentry which stores the name of the parent directory. The full path of each file is obtained by traversing this linked list of dentry structures. For non-regular files (e.g., sockets, terminals), we store additional data needed to restore them (not discussed further).

When restoring each process, we call chroot to restore the current root and chdir to restore the current working directory. Restoring open files requires calling functions that together implement the open and dup system calls. We do not use these system calls directly because our code has to be flexible enough to handle restoring shared data structures. For example, when the entire file descriptor table is shared between threads (or processes), once the table is setup for a thread, files do not need to be opened or duped in the second thread. To restore an open file, we first call a function that creates a file descriptor structure. Then we open the file using the flags, such as read/write, non-blocking I/O, etc., that were saved in the checkpoint. Next, we use the lseek system call to set the current file position. Then we dup the file descriptor so that it uses the correct descriptor number, and finally, we install this descriptor in the file descriptor table.

We handle temporary (or orphan) files by adding a temporary link to them during checkpointing and unlinking them after the restore. We ensure that all dirty file system buffers are committed to disk by calling the file-system wide sync operation as part of deferred kernel processing. As a result, we do not need to save and restore the contents of the file-system buffer cache.

D. Limitations

Our current implementation has several limitations that would need to addressed by a full implementation. Currently, we preallocate a fixed size of physically contiguous memory (default 100MB) for the checkpoint which limits the checkpoint size. Allocating discontiguous physical memory on demand would solve this problem, but we did not encounter any issue in practice because application checkpoints take only a small amount of memory, as shown in Section IV-C2. We have added support for a wide variety of kernel features to enable supporting many types of commonly used applications. Some of the feature that we do not implement currently include the EPOLL system call, pseudo terminals, message queues, Unix datagram sockets and the ability to pass file descriptors. The current implementation does not save and restore system or administrative applications like cron, udevd or getty and so these programs are shutdown and restarted normally on a kernel update. However, these programs do not have much state, and also do not require human intervention on restart. As more features are added to the implementation, these applications could be checkpointed as well. A process that is sleeping uninterruptibly may sleep indefinitely due to a kernel bug, which would stall quiescence. Such processes should be detected and not restored.

The most significant limitation of our current implementation is that it reinitializes devices on an update. Simply suspending and resuming a device does not work because the driver code may have been updated. Our approach works for simple devices such as network cards, disks, frame buffer, keyboard and mouse that can be quiesced or checkpointed easily. However, devices with significant state such as accelerated graphics devices will require support for updating drivers without reinitializing devices using techniques such as device driver recovery [6] and device virtualization [13], [7].

Data structure	Nr of fields	Nr of saved fields
vm_area_struct	16	7
mm_struct	51	5
task_struct	135	32
fs_struct	5	3
files_struct	7	1
file	18	10
sock	53	10
tcp_sock	76	48
unix_sock	13	10
pipe_inode_info	13	4
vc_data	82	12
fb_info	23	6
mousedev	18	7

Table II
KERNEL STRUCTURES AND CHECKPOINT FORMAT

IV. EVALUATION

We evaluate our system by analyzing our checkpointing format and code in terms of its suitability for supporting kernel updates. Then, we describe our experience with updating major releases of the kernel. Finally, we present performance numbers in terms of kernel update times and overhead.

A. *Code Analysis*

Table II provides a summary of our checkpoint format. There are a total of 13 data structures that are saved in the checkpoint. The table shows the number of fields in each data structure and the number of fields that we save from each data structure in the checkpoint. The saved fields include both primitive types or pointers to buffers that need to be saved. The rest of the fields are mostly implementation dependent and do not need to be saved.

The code consists of roughly 6,000 lines of code, as shown in Table III. Roughly 90% of the code resides in a separate kernel module, while the rest of the code is spread across various sub-systems. We separate kernel functions into four categories, based on how unlikely they are to be changed over time: system calls, exported functions, global functions and private functions. System calls are available to user applications and cannot be changed without breaking backwards compatibility and are thus the most stable. Exported functions are available to loadable modules and usually only have minor changes between kernel versions. Global and private functions are expected to change more frequently. Our checkpoint saving code uses 20 functions and all of them are exported. The restore code uses 131 functions, of which 5 are system calls, 93 are exported, 2 are global, and 31 are private.

We needed to use private functions for two purposes: 1) managing resource identifiers, and 2) performing low-level process creation and initialization. The kernel provides user threads with identifiers for kernel managed resources such as PID, file descriptors, port numbers, etc. When a process is restored, we must ensure that the same resources correspond to the same unique identifiers. However, since these identifiers are never modified, the kernel does not provide any exported or high-level functions to manipulate them. We believe that our solution is better suited for kernel updates because it doesn't impose any overhead for virtualizing identifiers during normal operation [10]. We also used private functions during process creation. The restore code is similar in functionality to the implementation of the execve system call that executes a file from disk. In our case, we create a process from the in-memory checkpoint data. The modifications needed were similar to the effort required to add support for a new executable format.

Subsystem	Lines of code
Checkpoint module	5257
Architecture specific	81
Memory management	70
File system	23
Process management	10
Networking	428
Total	5869

Table III
NEW OR MODIFIED LINES OF CODE

We had to modify some architecture-specific code to reserve memory during kernel boot. We also made some changes to memory management code to assign reserved memory to the restored processes. We also needed to make some changes to the Ext3 file system to prevent orphan file clean up on boot so that temporary files are not deleted. All changes to the networking code relate to adding progress tracking (See Section II-B) for reads and writes on TCP sockets, or changes needed to support TCP timestamps. Some of the changes do not alter the functionality of the kernel but were needed to provide access to previously private functions.

B. *Experience with Updating Kernels*

In this section, we describe the effort needed to use our system for performing kernel updates. We implemented our system starting with the Linux kernel version 2.6.28, released in December 2008, and have tested updating it, one major revision at a time, until version 2.6.34, released in May 2010 (roughly one and a half years of kernel updates). To ensure that our checkpoint-restart system works correctly, we have implemented a set of unit tests, where each test consists of a program that stresses a particular feature of the kernel. We verify that all the tests work correctly before and after an upgrade. After the units tests pass, we run several applications manually to ensure that they work correctly, and then more comprehensive benchmarks, as described in Section IV-C.

Table IV shows that on average, each revision consists of 1.4 million lines of added or modified code. There were six million lines of changed code over the six revisions in 23,000 files, including many data structure modifications. We updated our code from one version to the next using a typical porting procedure: 1) extract all our code from the kernel and keep it in a separate git branch, 2) merge our code into the next major kernel release using git merge functionality, 3) compile the kernel and fix any errors, 4) run an automated test suite that checks that all the features we have implemented are working correctly when updating between the kernel versions, and 5) commit the changes to our code so they can be used when moving to the next major release.

Table IV shows the number of lines that had to be changed manually for each kernel release. These changes are small, both compared to the number of lines changed in the major release, as well as the number of lines in the checkpoint code.

Kernel version	Lines of change in major release	Lines of change for checkpoint
2.6.29	1729913	42
2.6.30	1476895	16
2.6.31	1393049	7
2.6.32	1628415	5
2.6.33	1317386	50
2.6.34	882158	2

Table IV
SUMMARY OF UPDATES NEEDED FOR CHECKPOINT CODE

The majority of the changes were simple and were caught either during merge or during compilation. For example, several merge conflicts occurred when our code made a private function globally accessible, and some other nearby code was changed in the kernel update. Similarly, compilation errors occurred due to renaming. For example, we needed to use the TCP maximum segment size variable, and it was renamed from xmit_size_goal to xmit_size_goal_segs. These fixes are easy to make because they are caught by the compiler and do not the affect the behavior of the kernel or our code.

More complicated changes involved renaming functions and changing the function interface by adding arguments. In this case, we have to find out the new function name, and how to pass the new arguments to the function. For example, version 2.6.33 introduced a significant change to the interface used by the kernel to create and modify files and sockets. These changes were designed to allow calling file system and socket system calls cleanly from within the kernel. As a result, some internal functions used by our system were changed or removed, and our code needed to use the new file interface.

Our code needed to handle one significant data structure update conflict. Previously, the thread credentials, such as user_id and group_id, were stored in the thread's task_struct. In version 2.6.29, these credentials were moved into a separate structure, with the task_struct maintaining a pointer to this structure. We needed to change our system, similar to the rest of the kernel code, to correctly save and restore credentials. Two other data structure that we save, as shown in Table II, were updated, but they required us to simply pass an additional parameter to a function.

Finally, the most difficult changes were functional bugs, that were not caused by changes to data structures or interfaces. We found these bugs when running applications. We encountered two such issues with TCP code. Previously, a function called tcp_current_mss was used to calculate and update the TCP maximum segment size. In 2.6.31, this function was changed so that it only did a part of this calculation, and another function called tcp_send_mss was introduced that implemented the original tcp_current_mss behavior. Similarly, in 2.6.32, the TCP code added some conditions for setting the urgent flags in the TCP header, which indicates that out-of-band data is being sent. Our code was setting the urgent pointer to the value of 0, which in the new code set the urgent flag, thus corrupting TCP streams on restore. We needed to set the urgent flags based on the new conditions in the kernel.

All these ports were done by us within a day to a few days. We expect that kernel programmers would have found it much simpler to fix our code when updating their code. An interesting observation is that during porting, we did not have to change the format of the checkpoint for any of the kernel versions. As a consequence, it is possible to freely switch between any of these kernel versions in any order. For example, it is possible to upgrade to version 2.6.33 from version 2.6.28, and then go back to version 2.6.30 seamlessly. Our implementation allows updating kernels multiple times without additional overhead because we can reuse the same region of memory multiple times. We expect that developers will ensure checkpoint compatibility between consecutive major kernel releases, and believe that the effort to do further upgrades would be similar to the effort that we have described.

C. Performance

We conducted two types of experiments. First, we measured the throughput of server applications before and after the update. These experiments also show the downtime during an update. We used the Collectl system monitoring tool to measure throughput at the network interface level (sampled at one second interval). Second, we performed microbenchmarks to measure the per-process checkpoint size and time. All of our experiments run on the same machine with two Intel Xeon 3 GHz processors and 2GB of RAM, running Ubuntu 8.04 with our kernel that had support for updating the kernel.

1) Application Benchmarks: We tested several UDP (Quake game server, Murmer/Mumble voice-over-IP server) and TCP (MySQL, Memcached and Apache) server applications. Apache and Memcached used the EPOLL system call that our system does not support currently. Apache was compiled with EPOLL disabled and Memcached allows disabling EPOLL with environment variables. All these applications run after the update, without interrupting any requests in progress, and without requiring any other modifications. The Murmer and the Apache results are not shown because they were similar to the Quake and MySQL results.

a) Quake: We updated the kernel on the machine running the MVDSV 0.27 open source Quake server, while it was serving 8 ezQuake clients. The Quake server does not preserve its state across a reboot, and so the game needs to be restarted from the beginning. With our system, the clients resume after a short pause exactly where they were in the game before the update. For example, if a player was jumping when the update starts, the jump continues after the update.

The ezQuake client was modified to make it easier to compare reboot with seamless update. The unmodified client shuts down the current session and goes to the menu screen if the client stops receiving messages from the server for 60 seconds. With seamless update, this timeout is not an issue, but with a full reboot, the client would simply go back to the menu screen and then a user would have to manually reconnect to the server. To automate recovery for the reboot case, we modified the client so that it automatically attempts to reconnect to the server when no messages are received for 15 seconds. This change allows the client to reconnect to the rebooted server much faster and without any user interaction, making it easier to compare the systems. We chose the 15 second timeout because this time is much shorter than the time it takes for the server to reboot, and longer than the time

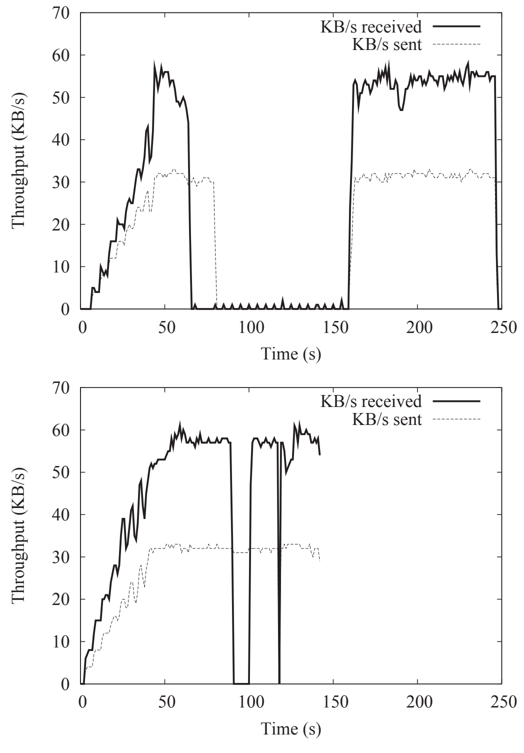


Figure 2. Quake reboot vs. update

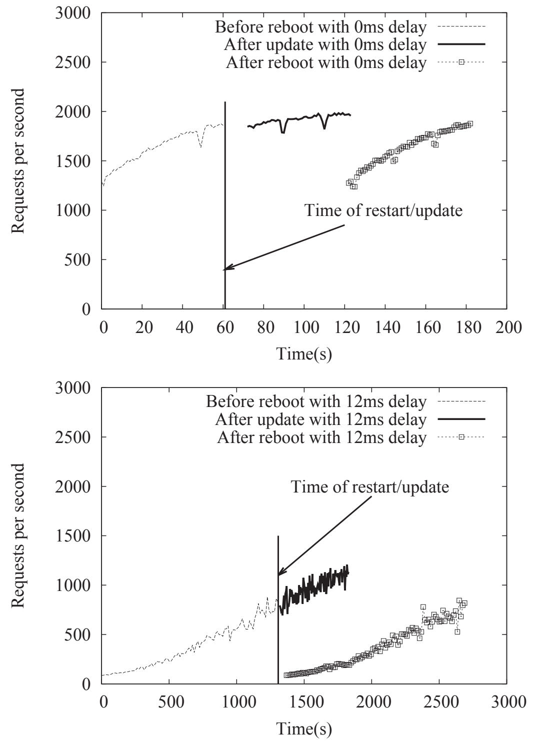


Figure 3. Memcached results with 0 ms and 12 ms database access times

it takes to update the kernel with our system. As a result, the client will reconnect as soon as the server is running again after the reboot, but it will not attempt to reconnect while the update is in progress.[3] Note that the game state is lost after reboot, but not with seamless updates.

Figure 2 shows the network throughput with reboot versus update. With reboot, the clients timeout after 15 seconds and then attempts to reconnect every 5 seconds as shown by the ticks at the bottom on the "sent" line. The server is down for roughly 90 seconds. With update, the server is operational in roughly 10 seconds, and all the clients resume normally.

b) MySQL: We used the sysbench OLTP benchmark to test the performance of MySQL server. This test involves starting transactions on the server and performing select and update queries in a transaction. The sysbench benchmark has no support for handling server failure. It returns an error when the MySQL server machine is rebooted and so we could not complete this test with a reboot. With our system, the kernel update time is roughly 10 seconds, similar to the Quake results shown in Figure 2. The output of the server drops to zero while the update is being performed, but the connection to the client is not dropped. Once the update is finished and the new kernel is started, MySQL continues to operate normally and the test runs to completion. There is no observable change in TCP throughput before and after the update, as observed at the sysbench client, and hence we do not show the MySQL throughput graph. No changes were required to MySQL or sysbench for this test.

c) Memcached: Memcached is a popular in-memory key-value caching system intended to speed up dynamic web applications. For example, it can be used to cache the results of database calls or page rendering. An application can typically survive a memcached server being shutdown, because it can recalculate the results from scratch and start using the cache when it becomes available again. However, cache contents are lost on a reboot, while our system preserves the cache and so the application can use its contents right after the update.

[3]We have not described our TCP checkpointing operation, but we do handle reconnection attempts during the update by simply dropping TCP packets, rather than sending TCP reset packets which would close the connection.

In this test, we compare the performance impact of the Memcached cache being lost after a reboot versus being preserved in our system. We generated 200,000 key-value pairs, consisting of 100 byte keys and 400 byte values. Then, we used a Pareto distribution to send requests to prime the cache, so that 20% of the keys from the 200,000 key-pairs make up 80% of all the requests. Then we simulate a web application that uses Memcached for caching the results of database queries. This application makes key lookup requests to the Memcached server. For each key, the application first makes a get request from the cache. If the key is found, it makes the next get request. If it is not found, then it waits for a short time (delay) to simulate calculating a result, and then issues a set request to store the result in the server, before making the next get request.

We use 12 ms for the delay value for a database access, which is the average time per transaction in our previous sysbench OLTP test. We also use 0 ms for the delay value to represent the best case, in which there is no cost for calculating a result. However, when a cache miss occurs, this instantaneous result still needs to be sent to the Memcached server, thus requiring a get and a set request.

In the first part of the experiment, we prime Memcached by sending it requests, and then in the second part we send a second set of request after doing a clean reboot or an update

Application	Quiescence time	Save state time	Restore state time	Checkpoint size
Quake	334 ms	99 ms	23 ms	135 KB
MySQL	338 ms	332 ms	75 ms	463 KB
Memcached	330 ms	6.4 ms	38 ms	112.3 KB

Table V
PER-APPLICATION CHECKPOINT TIME AND SIZE

Stage	Time
Initialize kernel	4.5 ± 0.3 s
Initialize services	6.9 ± 0.3 s

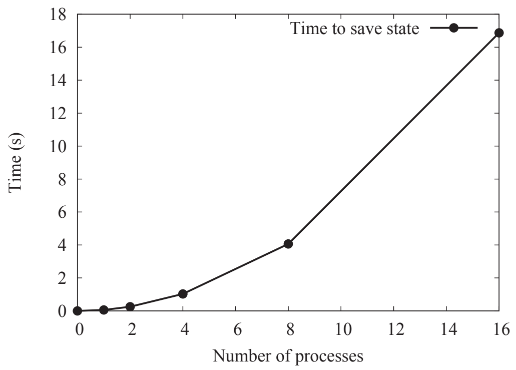
Table VI
KERNEL RESTART TIME

and compare performance in requests per second. The results of the experiment with 0 ms delay and 12 ms delay are shown in Figure 3. In the 0 ms case, we primed Memcached with 100,000 requests and then issued another set of 100,000 requests. In the 12 ms case, we primed Memcached with 500,000 requests and then made another 500,000 requests. The time where the Memcached server was rebooted or updated is shown with an arrow in both graphs.

The graphs show that after a reboot the number of requests per second declines because the contents of the cache are lost, and each miss adds extra overhead by restoring the lost cached value. In contrast when using our system the contents of the cache are preserved, which results in a smaller number of misses and the request rate stays at the same level as before the update. Performing a regular reboot loses the benefits of using in-memory caching until the contents of the cache are restored, while our approach preserves the performance benefit.

2) Microbenchmarks: Table V breaks down the time to reach quiescence, save the process state and restore the process state for the three applications described above. Quiescence time is measured from the time when we start the checkpoint process until we are ready to take the checkpoint (i.e., the last step in Section II-A) and includes the time to shutdown all the devices. The save state time is the time it takes to copy the kernel state into the checkpoint. The time to initialize the kernel is measured from when the new kernel's code starts executing to when the kernel starts the init process. The time to initialize services is measured from when the init process starts to when the saved processes begin to be restored, and the restore time is the time it takes for the saved applications to be restored and start running. The kernel quiescence time is roughly 330 milliseconds for each of these experiments. The checkpoint save time ranges from 6-350 ms, but as discussed below, we expect that this time can be reduced significantly with some simple optimizations. The checkpoint restore time has a smaller range from 25-55 ms. All these times are much lower than the time it takes to initialize the new kernel and the system services, as shown in Table VI. The last column of Table V shows the checkpoint size for each application, excluding the memory pages of the application.

We also conducted a microbenchmark to measure the checkpoint and restore time with increasing number of allocated frames in the system. The benchmark is run with 1, 2, 4, 8 and 16 processes. Each process in this benchmark allocates 16 MB of private memory using the mmap system call and writes to this memory to ensure that the kernel assigns page frames



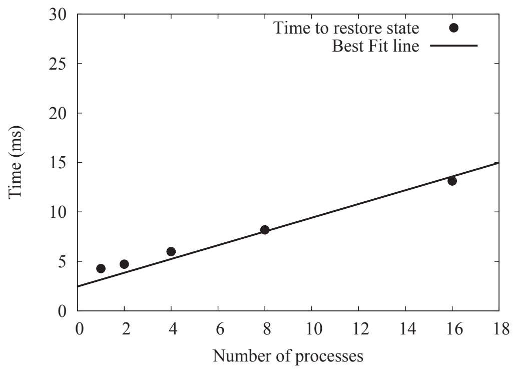


Figure 4. Mmap checkpoint-restore time

to the process. Figure 4 shows the checkpoint save and restore time with increasing number of processes. The save time is roughly one second per process in this benchmark with 16 processes and it grows with increasing number of processes because our implementation for saving state is not optimized. In particular, the code uses a linked list to detect shared pages. With 16 processes, and 16 MB of memory per process, there are 64K (2^{16}) pages in the linked list, making the search very slow (~2^{32} operations). These lookups can be sped up with a hash table or by using the reverse mapping information available in the memory manager to detect shared pages. The restore time per process is roughly one ms per 16MB process because we take advantage of the kernel memory manager to ensure that shared pages are assigned to each process correctly. Note that this time accounts for restoring the address space and is much smaller than the restore time for the benchmark applications (25-65 ms) which also need to restore other resources such as network buffers.

V. RELATED WORK

The Autopod system [10], [11] is closest to this work. It uses a checkpoint-restart mechanism, with a high-level checkpoint format, for migrating processes across machines running different kernel versions. Autopod uses a virtualization layer to decouple processes from their dependencies on the underlying operating system. Virtualization introduces performance overheads, and the virtualization layer itself needs to be maintained to keep up with kernel changes. This layer is not needed in our system because it is designed purely for kernel updates. The Autopod work suggests that applications are exposed to interrupted system calls, which makes applications susceptible to EINTR errors. By migrating state across machines that are already running operating systems, Autopod also does not seem to address quiescence issues such as caused by non-interruptible system calls, interrupts and quiescence of kernel threads. Compared to Autopod, we evaluate our system across major kernel updates and provide a detailed analysis of our checkpoint code and format, and all the code changes required for supporting kernel updates. Building on Autopod, Linux-CR [14] aims to add a general-purpose checkpoint-restart mechanism for applications to the Linux kernel, but it is also designed for migrating applications across machines.

Otherworld [15] is designed to recover from kernel failures by transferring the state of existing applications from the failed kernel to a secondary kernel. The design of Otherworld has inspired this work, but our aim is to ensure that kernel updates are performed reliably. Otherworld uses a best-effort resurrection process, because it does not have the liberty to achieve quiescence and does cannot restart system calls reliably. Also, we do not require any modifications to applications and evaluate the feasibility of the approach for kernel updates.

Several researchers have proposed dynamic patching at function granularity for applying kernel updates [1], [2]. None of these systems would work reliably when updating kernels across major versions because they require writing state-transfer functions for every updated data structure. CuriOS [16] recovers a failed service transparently in a microkernel OS by isolating and checkpointing the client state associated with the service. Virtual machines can be used to speed up reboot by running the existing and the updated kernel together [17]. Primary-backup schemes can be used for rolling kernel upgrades in high availability environments, but they require application-specific support [18].

VI. Conclusions

We have designed a reliable and practical kernel update system that checkpoints application-visible state, updates the kernel, and restores the application state. We have argued that this approach requires minimal programmer effort, no changes to applications, and can handle all backward compatible patches. Our system can transparently checkpoint the state of network connections, common hardware devices and user applications. It can achieve quiescence for any kernel update, and it restarts all system calls transparently to applications. We also performed a detailed analysis of the effort needed to support updates across major kernel releases, representing more than a year and a half of changes to the kernel. Our system required a small number changes to existing kernel code, and minimal effort to handle major kernel updates, consisting of millions of lines of code. Finally, we evaluated our implementation and showed that it works seamlessly for several, large applications, with no perceivable performance overhead, and reduces reboot times significantly.

Currently, our implementation resides entirely in the kernel and uses two monolithic system calls for saving and restoring checkpoint state. A more modular implementation would provide separate system calls for retrieving and restoring the checkpoint state for each kernel resource, allowing user-level code to implement the checkpoint algorithm more portably across kernel versions.

Acknowledgments

We would like to thank the anonymous reviews for their detailed feedback on the paper. We thanks William Ho, Colin Fung, David Carney, Eric Wright and Wael Louis for contributing to the code developed in this project. We also thank Michael Stumm and Angela Demke Brown and several other members of the SSRG group at University of Toronto for their feedback on initial versions of the paper. This research was supported by NSERC through the Discovery Grants program.

References

[1] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *Proceedings of the International Conference on Virtual Execution Environments*, 2006, pp. 35–44.

[2] J. Arnold and M. F. Kaashoek, "Ksplice: automatic rebootless kernel updates," in *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, 2009, pp. 187–198.

[3] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. 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 USENIX Technical Conference*, 2003, pp. 141–154.

[4] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: challenges and solutions to updating an operating system on the fly," in *Proceedings of the USENIX Technical Conference*, 2007, pp. 1–14.

[5] G. Kroah-Hartman, J. Corbet, and A. McPherson, "Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it," Linux Foundation, Dec. 2010, www.linuxfoundation.org/publications/whowriteslinux.pdf.

[6] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," in *Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2004.

[7] M. M. Swift, D. Martin-Guillerez, M. Annamalai, B. N. Bershad, and H. M. Levy, "Live update for device drivers," University of Wisconsin, Computer Sciences Technical Report CS-TR-2008-1634, Mar. 2008.

[8] F. L. V. Cao, "Reinitialization of devices after a soft-reboot," Usenix Linux Storage & Filesystem Workshop, Feb. 2007.

[9] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Engineering fault-tolerant tcp/ip servers using ft-tcp," in *Proceedings of the IEEE Dependable Systems and Networks (DSN)*, 2003.

[10] S. Potter and J. Nieh, "Reducing downtime due to system maintenance and upgrades," in *Proceedings of the USENIX Large Installation Systems Administration Conference*, 2005, pp. 47–62.

[11] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: a system for migrating computing environments," in *Proceedings of the Operating Systems Design and Implementation (OSDI)*, Dec. 2002, pp. 361–376.

[12] M. Siniavine, "Seemless kernel updates," Master's thesis, University of Toronto, Nov. 2012, http://hdl.handle.net/1807/33532.

[13] A. H. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "Vmm-independent graphics acceleration," in *Proceedings of the International Conference on Virtual Execution Environments*, 2007, pp. 33–43.

[14] O. Laadan and S. E. Hallyn, "Linux-CR: Transparent application checkpoint-restart in linux," in *Proceedings of the Linux Symposium*, 2010.

[15] A. Depoutovitch and M. Stumm, "Otherworld: giving applications a chance to survive os kernel crashes," in *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, 2010, pp. 181–194.

[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Curios: improving reliability through operating system structure," in *Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2008, pp. 59–72.

[17] D. E. Lowell, Y. Saito, and E. J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004, pp. 211–223.

[18] "Oracle database high availability features and products," http://docs.oracle.com/cd/B28359_01/server.111/b28281/hafeatures.htm.