Instant OS Updates via Userspace Checkpoint-and-Restart

Sanidhya Kashyap Changwoo Min Byoungyoung Lee Taesoo Kim Pavel Emelyanov[†]
Georgia Institute of Technology [†]*CRIU & Odin, Inc.*

Abstract

In recent years, operating systems have become increasingly complex and thus more prone to security and performance issues. Accordingly, system updates to address these issues have become more frequently available and increasingly important. To complete such updates, users must reboot their systems, resulting in unavoidable downtime and further loss of the states of running applications.

We present KUP, a practical OS update mechanism that employs a userspace checkpoint-and-restart mechanism, which uses an optimized data structure for checkpointing on disk as well as a memory persistence mechanism across the update, coupled with a fast in-place kernel switch. This allows for instant kernel updates spanning across major kernel versions without any kernel modifications.

Our evaluation shows that KUP can support any type of real kernel patches (e.g., security, minor or even major releases) with large-scale applications that include `memcached`, `mysql`, or in the middle of the Linux kernel compilation, unlike well-known dynamic hot-patching techniques (e.g., `ksplice`). Not only that, KUP can update a running Linux kernel in 3 seconds (overall downtime) without losing 32 GB of `memcached` data from kernel version v3.17-rc7 to v4.1.

1 Introduction

Today, computer users routinely update their operating systems either to patch security vulnerabilities, fix bugs, or add a new set of features to the existing system. Unfortunately, to fully incorporate a new update, users have no choice but to restart their systems, which results in the loss of the running states of applications. This unavoidable disruption not only brings inconvenience to end users, but also causes an adverse financial impact on business.For example, updating a `memcached` [57] server at Facebook that caches around 120 GB data in 144 GB RAM [24] requires a prolonged warm-up phase of 90-100 minutes [30]. The more critical problem is when a system update fails. Such failure often leads to additional downtime or maintenance costs thereafter, resulting in an immediate loss of active customers [20, 21]. Still, even with the unavoidable disruption and risks involved in update failures, system updates are necessary to promptly mitigate the known security issues and resolve critical bugs that might hurt the correctness of operations [8, 25, 44, 68, 71].

To solve these problems, two large sets of techniques are used in practice: dynamic hot-patching and rolling updates. Dynamic hot-patching (or live update) [4, 67,

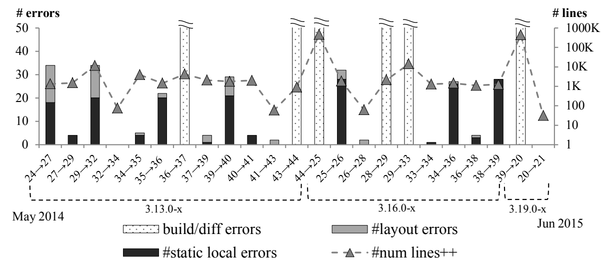


Figure 1: Limitation of dynamic kernel hot-patching using `kpatch`. Only two successful updates (3.13.0.32→34 and 3.19.0.20→21) out of 23 Ubuntu kernel package releases. X-axis represents before-version and after-version, and dotted bars represent failures in executing `kpatch`. Each failure case represents the following legends: build/diff errors—`kpatch` failed during a build and diff processes; # static local—a patch modifies a value of static local variables; # layout error—a patch changes a layout of a data structure.

73] directly applies patches to the running kernel. As a result, system updates can be performed without incurring application downtime. However, dynamic hot-patching is inherently limited to patches that do not semantically modify data structures. Therefore, it is commonly used to apply simple security patches that contain minor code changes. For instance, `kpatch` [67], a popular dynamic update tool developed by Red Hat, was able to support 2 out of 23 minor updates over a year of Ubuntu's kernel releases (see Figure 1).

Rolling updates [20, 60] are another viable technique for large-scale systems. In rolling updates, system administrators apply an update to a small group of machines; if there is no failure, they apply the same update to the rest of the machines in the data center. This helps administrators minimize the risk of update failures and system downtime. However, this process requires careful planning to minimize the disruption of running services due to the inevitable downtime of applications.

There are several research projects that attempt to solve the OS update problem either by proposing new OS construction [5, 7, 23, 65, 80], or by providing a transfer function for system updates [28]. In order to keep their internal states switchable, such OSes require radical design changes, thereby making these modifications impractical for the commodity OSes [19, 69].

To address these issues, we present KUP—a new update mechanism that allows for prompt kernel updates without modifying a commodity OS. KUP incorporates application checkpoint-and-restart (C/R) for saving and restoring the application states before and after the update,

and an in-place kernel switch to quickly boot into a new kernel. This allows KUP to minimize the downtime of running applications during the system update. Unlike dynamic hot-patching mechanisms, KUP can perform a full system update regardless of the complexity of updates, and thus can support a broader range of system updates, including security and bug fixes, the addition of new features, and even major kernel upgrades. To address update failures, KUP provides a safe fallback mechanism that provides a mechanism to restore the original system, checkpointed right before the update.

KUP works by leveraging the OS provided infrastructure such as `procfs`, `ptrace` system call [37] for obtaining information during the checkpoint of an application, and process forking (`clone`, `fork`) mechanism along with other system calls [15, 37] and `netlink` sockets to restore the application after update. To enable an efficient, userspace-based C/R of an application during an update, we design an efficient checkpointing storage format that improves both the checkpoint size and KUP's performance. However, the aforementioned solution of KUP still ends up storing RAM data on the disk, which is not suitable for disk-less systems [30, 57].[1] We further extend KUP and make it generic enough to support disk-less systems by proposing a memory persistence mechanism, which we implement via binary patching and dynamic OS instrumentation. This approach is complementary to our userspace solution.

This paper makes the following three contributions:

- We design a simple, yet robust update mechanism by using application C/R, and implement an open source prototype of KUP.
- We show KUP's effectiveness by providing an in-depth analysis of the proposed techniques with real applications, micro-benchmarks and full software stack representing generic data center application.
- We devise, implement and evaluate a safe fallback mechanism for KUP that can be easily realized through application C/R and an in-place kernel switch.

The rest of this paper is organized as follows. §2 compares KUP with previous work. §3 gives the high-level ideas of KUP's approach to instant kernel update. §4 points out the challenges and §5 describes our design. §6 explains our implementation and §7 evaluates KUP's performance. Lastly, §8 discusses its limitations and potential optimizations and §9 concludes.

[1] The `memcached` servers at Facebook are disk-less and they keep everything in memory.

2 Related work

In this section, we compare KUP's approach to previous studies in four areas: dynamic hot-patching, new OS designs, live migration, and application C/R for updates.

Dynamic hot-patching. Industries rely on solutions such as `Ksplice` [4], `kpatch` [67], and `KGraft` [73] to promptly mitigate the security vulnerabilities without any significant outage of their production systems. However, as shown in Figure 1, the hot-patching techniques have critical limitations, especially in handling data layout changes, thus making it impossible to guarantee the safety of live updates. Previous studies have proposed dynamic software update schemes on event-driven systems [1, 27, 28, 36], object oriented languages [39, 63] and even C language [4, 35, 53, 54, 56] including formal proofs [34, 55, 72].

However, unlike previous update techniques, KUP relies on a whole kernel switch to apply any kind of patch, regardless of its complexity. KUP allows for system-wide live updates without modifying programs or tracking their state changes, unlike previous works [19, 35].

New operating system designs for live update. Another approach is to design a new OS with well-defined abstractions between interfaces and implementations so that each component can be replaced online without disruption. Representative examples include Microkernel (e.g., exokernel [23], K42 [5, 7, 70], Barrelfish [6, 80]) and LibOS (e.g., Drawbridge [65]). In particular, Proteos [28] based on MINIX 3, and a Linux variant [69], are designed to update their components online and also transform internal data structures to adopt the updated kernel. Unfortunately, with a large amount of code changes (e.g., new features), constructing the state transfer functions either requires manual effort or is infeasible in many cases [5, 7, 11, 28, 69, 70].

Autopod [66] and the work by Siniavine et al. [69] are closest to KUP. Autopod uses a namespace mechanism for process decoupling and migrating applications across machines, whereas Siniavine et al. provides a kernel space C/R mechanism coupled with an in-place kernel switch to update the OS, and employs techniques similar to those of Otherworld [19]. The work by Siniavine et al. heavily modifies various kernel subcomponents to checkpoint and restore the applications entirely in the kernel space without any user intervention. In contrast, Otherworld tries to recover applications from the kernel failures by transferring them to the new kernel via an in-place kernel switch.

Unlike these studies, KUP achieves its goal of instantly updating the systems via userspace C/R and an optimized in-place kernel switch *without modifying the kernel source*. KUP faces a different set of challenges, such as (a) how to efficiently checkpoint storage data structure for

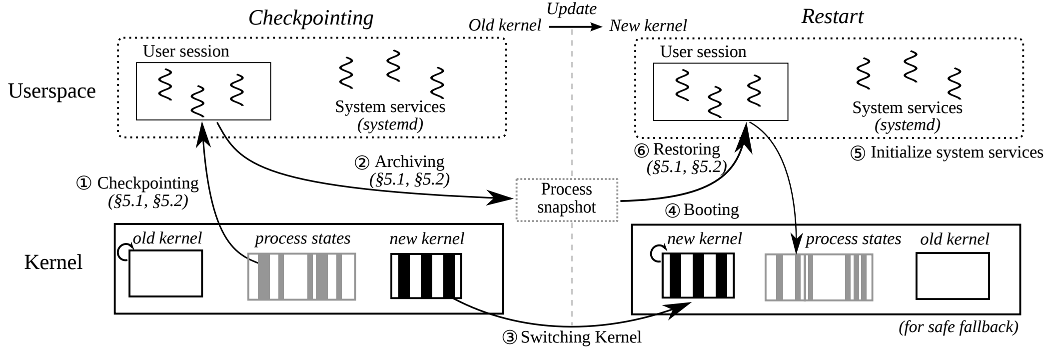


Figure 2: Overview of KUP's updating procedures. KUP first checkpoints user's processes ①, and archives their snapshots ②. After checkpointing selected processes in a user's current session, KUP replaces the old kernel to the new kernel image ③, and finally switches to the new kernel ④. After the new kernel boots, KUP first initializes its system daemons ⑤, and finally restores snapshots of user applications ⑥.

effectively using both checkpoint and restore techniques, and (b) how to implicitly modify the kernel functionality to remove the redundant memory copy phase from the application C/R, which we do via a binary patching mechanism. KUP leverages the kernel-exposed userspace information to extract the kernel-resident data-structures' data via `procfs`, netlink sockets and system calls [15]. This makes the process checkpointing independent of both the kernel update and the checkpoint format.

Application C/R. The area of application C/R has been explored in various contexts: object-oriented languages [50, 77], single process [61, 64], multiple processes [2, 49], even across multiple computers in a distributed environment [43, 79]. It has also been explored solely for an optimized restoration scheme [81]. In practice, application C/R has been used for bootstrapping system startup [51], achieving fault-tolerance and performance in HPC [26, 33], debugging [61, 77], and even for recovering system integrity [45–47].

Unlike these approaches, KUP can retrofit the known application C/R scheme for live updates. Currently, it relies on `criu` [22] for application C/R because of its mature code base and stable checkpointing mechanism without modifying the kernel. General application C/R approaches focus on reducing the quiescence period, but KUP suffers from a memory snapshotting bottleneck, since the kernel provides the quiescence mechanism via the `ptrace` system call [37] for KUP to leverage. KUP uses incremental checkpoint [64] and on-demand restore [58] to reduce the downtime during application C/R. Unlike libckpt [64], KUP does not require any modifications to applications and uses a new file format to effectively merge both techniques.

Live update with virtual machine migration. The internal data structures of operating systems can be captured by providing underlying abstractions such as virtual machine [12, 38] or namespace containers [41, 48]. Microvisor [52] allows for both live update and rolling update of host and guest operating systems by migrating a suspended virtual machine to another host before proceeding with an update. On the other hand, ShadowReboot [78] updates the guest OS by forking and restoring the snapshot of the old guest state on the same host.

Unlike these approaches, KUP directly enables the in-place live update of an OS without relying on a separate host. This allows KUP to be easily used on clusters with heterogeneous hardware, where live migration is problematic due to the incompatibility of CPUs and devices.

3 KUP Life Cycle

KUP's goal is to update an OS (e.g., kernel and system services) by immediately applying patches without incurring observable downtime. KUP achieves this goal by first preserving the states of running applications and then restoring their previous states on the updated OS. This will allow administrators to update systems with minimal downtime. Importantly, this update procedure can benefit end users, by allowing for frequent updates not only for immediate security fixes, but also for less critical changes, such as performance improvements and the incorporation of new functionality (e.g., new I/O scheduling policy and also software rejuvenation at the OS level [40]).

Conceptually, KUP's update procedure consists of three steps. It first checkpoints the running applications, then switches to the updated OS, and finally restores the suspended applications. This section describes KUP's update procedure followed by a scenario enabled by KUP (i.e., safe fallback). Figure 2 gives a holistic overview of KUP's update cycle for applications.

Checkpointing applications. Before updating the OS, KUP takes a snapshot of running applications. This snapshot is made up of process states, consisting of their memory space (e.g., code/data sections and stack/heap, etc.) and their internal states in the kernel, including for example the states of sockets, etc. (① checkpointing in Figure 2). The snapshot is then stored in a persistent stor-

age, where KUP can fetch it after switching to the updated OS (② archiving).

Switching kernel. After checkpointing, KUP first loads the new kernel binary to its memory. Then, it switches its running kernel (old version) to the new version that was just loaded (③ switching kernel). The new kernel quickly boots up by skipping most of the hardware initialization routine (such as slow BIOS and POST) as the devices and peripherals were already initialized ④. The system management daemon relaunches the required system services after the kernel initialization ⑤. KUP never checkpoints the system services, since they are already maintained by the system management daemon.

Restoring applications. After running the newly updated kernel along with its system services, KUP starts the process of restoring the user's applications that were suspended before the update. It reconstructs the process-specific states inside the kernel and then restarts the application (⑥ restoring).

The application C/R and kernel switch not only allow for a seamless kernel update, but also facilitate a mechanism to safely fall back to the previous kernel in case of an update failure.

Safe fallback. KUP also supports safe fallback, a mechanism that allows for automatic fallback to the previous OS state. With this mechanism, KUP tries to preserves the availability of the older kernel to tackle failure cases introduced by a buggy updated kernel as well as the liveness of the applications.

4 Challenges

The existing general C/R tools can efficiently checkpoint and resurrect applications provided by the kernel-exposed userspace information due to the inclusion of a namespace-based container mechanism [14, 32, 41]. However, they suffer from two well known issues: *quiescence* and *memory checkpointing*. Quiescence is a time span in which all the kernel-resident data structures and the states of threads at the kernel and user levels are consistent with the checkpointed image. These both lead to application downtime during C/R. In our experiments, the quiescence time period constitutes only 0.01%–2% of the total downtime while updating the OS with KUP, whereas the major downtime occurs because of the memory dump (95%–99%). Hence, we focus on understanding the impact of the memory dump during the update as well as the inherent limitation of the userspace C/R from an OS update perspective. Later, we will briefly discuss how KUP achieves quiescent state in §6 and its contribution to the downtime.

Memory serialization for application C/R. We will first give a brief background of the existing state-of-the-art C/R techniques which KUP use:

- **Incremental checkpointing.** Checkpointing the memory of processes imposes significant downtime when using existing userspace application C/R tools. The downtime will be unacceptable for applications like `memcached`, since for these it is preferable for the application's data to be persistent and usable after an update. To resolve this issue, the C/R component of KUP implements *incremental* checkpointing. This idea itself is not entirely new [12, 64], but it reshapes this as an application-agnostic userspace C/R mechanism: it takes multiple *asynchronous* snapshots of the process's memory, followed by a *synchronous* one. This leads to minimal downtime. KUP only suspends the process in the last iteration, because of the minimal checkpointed data in the last *asynchronous* iteration.
- **On-demand restore.** The naive restore mechanism of the existing C/R tools imposes equivalent downtime, similar to the naive checkpoint mechanism. For example, sequentially reading the checkpointed data from the disk imposes an unacceptable downtime for the memory intensive applications, like `memcached`, during the restore phase. To resolve this issue, the C/R component of KUP starts the process without loading its entire memory contents, instead it reloads them *on-demand* when the process tries to access a particular memory address. This accelerates the process restart, hence decreasing the application's downtime during restore.

However, the coupling of both on-demand restore and incremental checkpoint degrades the overall performance after an update. In particular, this coupling suffers from two problems: (1) the overhead of finding a corresponding file page when a page fault occurs in a checkpointed snapshot image, and (2) the overhead incurred by `mmap()` when binding each reloaded page in userspace. Note that performance and scalability problems of `mmap` operations in the Linux kernel are well known [13].

Eliminating redundant memory copy. Even though the merging of the aforementioned C/R techniques speeds up the C/R process, it copies the memory twice between the kernel and the user space. Furthermore, the backend storage, which keeps the snapshot persistent across system updates, becomes the bottleneck for disk-less systems. For example, it is not possible to update Facebook's disk-less `memcached` servers, since they cache 120 GB of data in memory and do not have any storage medium available for the warm-up phase.

5 KUP Design

KUP's design goals are to (1) instantly update a running kernel, (2) avoid any observable downtime, (3) ensure no kernel modification, and (4) supporting all types of

patches, unlike previous approaches [19, 69]. To realize these goals while addressing the aforementioned challenges, we first present a new data structure called FOAM to efficiently serialize memory for both C/R techniques (§5.1). Then we present another optimization technique, called *persistent physical pages* (PPP), to further decrease the downtime during kernel update (§5.2). Later, we introduce KUP's safe fallback mechanism.

5.1 Userspace Application C/R

By using incremental checkpoint, the C/R component of KUP suffers from slower restoration than the naive checkpointing. This results in slower application restart due to its extra work in maintaining a sequence of images taken during the incremental checkpoint. Moreover, the performance further degrades when the incremental checkpointing is combined with on-demand restore. For example, depending on the application's writable working set, memory address ranges (e.g., dirty pages) as well as their respective metadata (e.g., previous page and address region) get fragmented and scattered across multiple checkpointed dumps (files). Hence, the C/R component needs to install a huge number of mappings to backup *each page* by the checkpointed dump in the restore phase. This imposes non-negligible overhead for enabling the on-demand restore with the incremental checkpoint, and also results in lots of context switches while installing the mappings at the page granularity.

To solve this problem, we design a simple yet effective data structure called *file offset-based address mapping* (FOAM), for quickly restoring the processes' pages from the checkpointed images. FOAM uses a direct one-to-one mapping between the process address and file offset in a *sparse* file. Due to this, KUP avoids the maintenance cost of file indexes and fine-grained pages. Instead, KUP can bind the checkpointed image to the whole address space.

We prepare this sparse file large enough to represent 64-bit virtual address spaces ($2^{48-1} = 128$ TB excluding kernel and canonical address spaces). We leverage the concept of *holes*, which all modern filesystems support, such that a single file is large enough to express whole virtual address spaces (e.g., 16 EB support by xfs and btrfs)[2] while never occupying such a huge amount of physical storage spaces underneath.

FOAM's approach brings three advantages to KUP for userspace-based OS update: (1) it eliminates the overhead to maintain metadata by directly mapping process address space to the file offset (e.g., mapping virtual memory region at page granularity); (2) it resolves the data fragmentation issue from the incremental checkpointing (e.g., new snapshot might contain a newly updated memory region if its size is changed from its previous snapshot); (3) it simplifies the design to enable the on-demand restore (e.g., mapping a single file to the entire process space). With FOAM, KUP can leverage the incremental checkpoint and on-demand restore together, thereby reducing the downtime caused by application C/R.

[2] FOAM is not limited by a file system that does not support such a huge filesize. For example, to support ext4, KUP can serialize virtual address spaces into 8 files for each process as ext4 supports 16 TB file size.

5.2 Reusing Persisted Memory Across Update

Even though the FOAM approach is able to decrease the downtime experienced by application by using state-of-the-art C/R techniques, it still suffers from redundant copying of the process's memory before and after the update. Also, there should be sufficient space to save the checkpointed data. To thwart these problems, KUP introduces a new mechanism called *persistent physical pages* (PPP). PPP removes the redundant copying of the memory from the kernel to the userspace and vice-versa. Instead, it preserves the process' memory across the system update.

PPP first saves the virtual address of a process and its corresponding physical mapping at the page granularity. Then, while booting the new kernel, KUP reserves the mapping information and the corresponding pages to forbid their usage. Later, KUP relies on the page fault handler to rebind the pages with the restoring application in an on-demand fashion. This allows KUP to reuse the same physical pages at the time of restoration after the update. This technique is effective in two ways: (1) it avoids redundant copies of process' memory during application C/R, rather preserving them without extra overhead; (2) it allows for an instant, page-level on-demand restoration of a process. Unlike any application C/R, PPP does not degrade the performance of the restored process, as it quickly rebinds the physical page upon a page fault.

Currently, the kernel does not provide any functionality to completely implement PPP in the userspace. The previous work [69] has heavily modified multiple kernel subcomponents to achieve it entirely in the kernel space. On the contrary, keeping KUP's design goals in mind, we instead hook the kernel's memory management unit with the binary patching mechanisms, such that PPP is easily applicable to the current commodity OSes.

To implement PPP, KUP needs to reserve the set of pages as well as the virtual-to-physical mapping information after the kernel update and then handle the page faults for the restoring process. Since both steps are only possible in the kernel space, we use static binary instrumentation for injecting the code in the kernel binary image and dynamic kernel instrumentation to hook the page fault handling functionality. Next, we describe binary patching in detail, and later the steps to perform PPP.

Binary patching. For memory reservation of pages and their mappings, KUP performs binary patching on the new kernel binary in which the system will boot into. Before starting the checkpointing process, KUP first does the

static binary instrumentation on the kernel image. Then, it adds a new section to the binary. This new section contains the memory reservation code. Later, KUP finds a specific function where the memory reservation code should run first. Then KUP hooks that function and injects the memory reservation function's address, so that this function can run before the specified function. After its execution, the code jumps back to the specified function and resumes the normal kernel execution.

To handle the page faults of the restored process, KUP relies on dynamic kernel instrumentation by hooking the kernel's page fault handling function. KUP executes its own page fault handling code, which binds the page corresponding to the stored virtual-to-physical mapping of the restored process.

Hence, by using both instrumentation techniques, we believe that KUP's PPP is easily applicable to commodity OSes while strictly adhering to the design goals of KUP.

PPP in action. KUP enables PPP with the following five concrete steps:

(1) Before checkpointing, KUP performs binary patching on the new kernel image.
(2) During checkpointing, the virtual-to-physical mapping of the targeted application is passed to a kernel module that reserves a memory space where the per-process page mapping information is saved.
(3) During new kernel's reboot, with the binary patched module, KUP globally reserves the set of per-process requested pages passed on from the previous kernel.
(4) Before application restoration, KUP patches the page fault handler with its own version to specially handle the page faults from the restored application.
(5) During restore, KUP only restores the states of the process except the memory pages and then restarts the application.

Steps (1), (2), and (4) implicitly hook the kernel. In step (1), we patch the new kernel image with our memory reservation code. In step (2), we use a kernel module for saving the per-process mapping information passed to it during the checkpoint phase. In step (4), KUP dynamically patches the page fault handler to specially consider the page faults from the restored process. After handling all the page faults, KUP falls back to the original function, thereby not imposing any overhead after the restoration.

By using PPP, KUP can dramatically reduce the total downtime with respect to the application C/R. Unlike dynamic hot-patching, PPP is applicable across multiple kernel updates, since it does not modify any in-kernel data structures. Instead PPP leverages the stable in-kernel functionality for the memory reservation and page fault handling. PPP does not introduce any security issues as the superuser is only responsible for the whole update procedure. Further, PPP's approach inherently adopts on-demand restore, as it naturally rebinds the faulted page to the virtual address, whereas FOAM explicitly binds the process's address to an image-file backed file descriptor to enable on-demand restore.

Component	Lines of code
`criu` / on-demand restore	810 lines of C
`criu` / FOAM	950 lines of C
`criu` / PPP	600 lines of C
KUP `systemd, init`	1040 lines of Python/Bash
`criu` / others, `kexec()`, etc.	150 lines of C
Total	3,550 lines of code

Table 1: An implementation complexity (lines of code) of KUP.

5.3 Safe Fallback

After updating the kernel, a system or an application may not run correctly for many different reasons. For example, an application may not run correctly due to the updated buggy file system [9] or a buggy system configuration [59]. In such cases, KUP's key ability can be retrofitted to support safe fallback, i.e., by carrying out instant downgrade. To allow the OS and applications to recover from failure, KUP takes exactly the same steps, but with the previous kernel image.

In particular, before switching kernels, KUP loads the safe fallback kernel image in a reversed memory section. Then, if the failure occurs, KUP performs the instant downgrade using this safe fallback kernel image. Depending on where such a fault occurs, this downgrade can be triggered because of a kernel layer or an application layer. KUP supports both cases: (1) if a fault is detected at the kernel layer (i.e., a kernel panic), KUP switches to the previous kernel; (2) if a fault is detected at the application layer (i.e., an application restoration fails), KUP switches into the previous kernel with the help of the system management daemon. Note that policies to detect these update failures are orthogonal to KUP's approach, as one can easily plugin any custom policy into KUP.

6 Implementation

We implemented our prototype of KUP on Linux v3.17 by using `criu` (v1.4) [22] for application C/R and used `kexec()` [31] for the in-place kernel switch. We modified various components of `criu` and implemented userspace binary patching module in python along with the pluggable kernel modules for PPP. The whole code consists of 3,550 lines in total (see Table 1).

FOAM. Besides application C/R, KUP also relies on `criu` for achieving quiescence, which is obtained as soon as the kernel pulls the process out of the sleeping state. `criu` uses the `ptrace` [37] functionality to achieve this. This ensures that KUP can only begin the application checkpointing when the process is back in the userspace as the kernel notifies the KUP process. Moreover, KUP concen-

Date	Ubuntu minor update From → To	Updatable? KUP	kpatch	Analysis S	D	Patch complexity #files	+#lines	-#lines	#Bugs (security)	Example (a reason for failure)
2014/05/15	Linux 3.13.0-24 → 27	✓	-	18	16	150	1,346	1,480	139 (5)	New `.is_fw_header` in `rtl_has_ops` struct
2014/06/04	Linux 3.13.0-27 → 29	✓	-	4	0	178	1,495	984	168 (4)	New static var. `pirq_ite_set` in `pirqmap()`
2014/07/15	Linux 3.13.0-29 → 32	✓	-	20	14	551	11,890	5,694	756 (7)	New `.cache_events` in `power_pmu` struct
2015/08/13	Linux 3.13.0-32 → 34	✓	✓	0	0	6	77	18	10 (4)	CVE-2014-4943 is fixed.
2015/06/14	Linux 3.19.0-20 → 21	✓	✓	0	0	4	31	2	1 (1)	CVE-2015-1328 is fixed.

Table 2: Snippets of our analysis on minor updates spanning twelve months, in the Ubuntu distribution, using KUP and `kpatch` (refer Figure 1 for a full picture). Unlike `kpatch` failed to support 21 cases out of 23, KUP was able to support all 23 cases (checks under *Updatable?*). To understand why `kpatch` cannot support such updates, we analyzed each minor release to see (1) how many new *static variables* (*S* under *Analysis* column) are introduced, and (2) how many *data structure layout* (*D* under *Analysis*) changes are included. To be specific, we also included an example of such cases that `kpatch` cannot easily support, even with manually designed update payload.

trates on per-application C/R that only saves selected applications during an update. During restore, `criu` restarts the process with the help of parasite, thereby allowing the application to resume from where it was suspended after reconstructing all of its states.

To enable on-demand restoration, we `mmap()` process' virtual pages with the checkpointed image. For this, we modify the `criu`'s *parasite* code, an injected process coordinates process restoration, to correctly back the memory pages by using the proper offset from the image file.

We implement `FOAM` by modifying both the incremental checkpoint implementation and the basic restore implementation of `criu`. For `FOAM`, we use `xfs` as KUP's backend file system to store the checkpointed image as `xfs` supports a filesize up to 16 TB. Otherwise, this technique can be generally applied by serializing the virtual address spaces to multiple image files (see §5.1). KUP maps the entire virtual address space into file offsets and then creates *holes* by truncating it to 2^{48-1} bytes.

`PPP`. We implement `PPP` by hooking the memory subsystem via dynamic kernel instrumentation and static instrumentation. Before rebooting the newer kernel, we perform binary patching on the kernel binary (`vmlinuz`) by hooking the `setup_arch()` function for memory and page reservation for the restoring process. The `setup_arch()` function is one of the oldest functions existing in the kernel which we hook. While checkpointing, a kernel module obtains the virtual address-to-page frame number mapping from `criu` (via `ioctl`) and reserves the information in a global memory and marks the pages to be unusable. At the time of reboot, the hooked function reserves the process' pages and the global memory with the help of the boot allocator. Later, the page fault handler refers the global memory for accessing the relevant pages corresponding to the restoring process. During restore, KUP uses `jprobes` to achieve the on-demand page fault handling by hooking the `handle_mm_fault()`, which updates the page table entry of the restoring process after binding the page to the corresponding faulted address. We do not modify any in-kernel data-structures for `PPP`.

Optimizing `kexec()`. Unlike dynamic hot-patching techniques that employ unstable in-place updates (see Figure 1), KUP seeks a robust way to replace the entire kernel, thus allowing a complete switching between two different kernel versions. Instead of relying on hard or soft reboot, we use `kexec()`, which is originally designed to debug a crashed kernel with a secondary kernel image in a post-mortem manner. `kexec()` is robust as it supports all minor updates of Ubuntu from May 2014 to July 2015, and is over 50% faster than soft reboot (see Table 3). `kexec()` acts as a minimal bootloader. It is responsible for resetting device states before booting into a new kernel.

We make two optimizations to the stock `kexec()` to reduce the booting time: (1) avoid polling of `PCIe` slots, if they are not used before switching the kernel; (2) bring CPUs *lazily* online during boot, since we observed that the synchronization among system services slows down the system's boot.

7 Evaluation

We evaluate KUP by answering the following questions:

- How effectively can KUP apply patches compared to popular dynamic hot-patching techniques? (§7.1)
- How much downtime does KUP incur while running various types of applications during updates? (§7.2, §7.3)
- How effective is each technique in decreasing the system downtime during updates? (§7.4)
- How effective is KUP in a full software stack composed of multiple inter-related applications? (§7.5)

7.1 Dynamic Hot-patching vs. KUP

We compare KUP with `kpatch` (latest version—v0.2.2) to show its effectiveness when switching between various versions of Ubuntu kernel releases. Table 2 shows in-depth results on five specific patches out of 23 patch cases discussed in Figure 1. Overall, KUP is able to support all 23 updates, whereas `kpatch` fails to update 21 versions except two. It fails if any of the following happens in the patch: (1) build and diff failure for the kernel versions; (2)

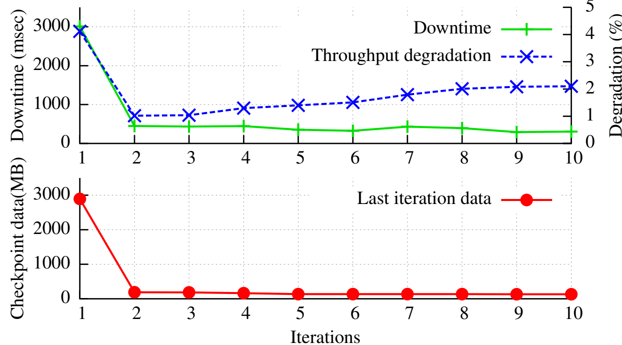


Figure 3: Throughput degradation, downtime and the data dumped in the last iteration measured for varying iterations during incremental checkpoint of `memcached`.

a patch modifies a value of static local variables (shown as *S* in *Analysis* column); (3) a patch changes the layout of a data structure (shown as *D* in *Analysis* column).

The first case, build and diff failure, is an implementation bug of `kpatch` and is not related to the inherent limitations of dynamic hot-patching techniques. We evaluated 23 patches and `kpatch` fails for 11 such cases. The second case consisting of the update of static local variables' values, causes a conflict with its runtime value semantics after hot-13 patches belong to this category that can be handled via kernel module. The last case entails a layout change in a data structure, in which `kpatch` has to locate and update all of its instances during runtime to safely update the system, but completely identifying them is far beyond the current state-of-the-art techniques and involves many challenging problems in the domain of program analysis [17]. `kpatch` fails on 11 patches for this case.

The aforementioned causes can be related to the patch complexity, which we measure in terms of the number of files and lines of code changed (*Patch complexity* column) and the number of bugs addressed (*#Bugs* column). The kernel updates, 3.13.0-32 → 34 and 3.19.0-20 → 21, are the only two updates that `kpatch` can handle, as there are relatively few changes in the code: the former changes 6 files and 71 lines of code to fix 10 bugs, and the latter one modifies 77 files and 31 lines of code to fix one bug. All other updates alter more than 150 files and 1346 lines of code, thereby resolving 139 bugs.

7.2 Applying KUP with Running Applications

In this section, we evaluate KUP with various types of real applications and provide our analysis of the impact of KUP on their performance across a system update. KUP is able to handle all major updates from v3.17-rc6 to v4.1. We perform all of our experiments on a 4 core machine with 64 GB RAM.

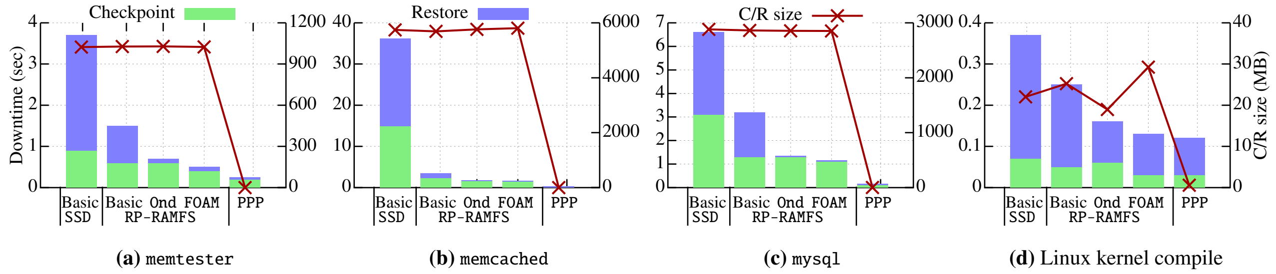
Faster storage medium: `RP-RAMFS`. As discussed in §5, the backend storage medium can inhibit the end-to-end performance of KUP. Users can circumvent this by adopting a RAM-based file system (reboot-persistent RAM file system—`RP-RAMFS` [18]) that stores its data purely in RAM but makes it persistent across system reboots. Note that `RP-RAMFS` is complementary to any C/R, even though it can dramatically decrease the application downtime. We use `RP-RAMFS` to show that users can use the emerging persistent memory [62] as a persistent-across reboot memory for the application C/R. However, `RP-RAMFS` has one fundamental limitation: the system needs enough free RAM to store the snapshot.

Targeted applications. While KUP can support most of applications (see §8 for limitations), we select four applications, namely `memtester` [10], `memcached` [42], `mysql` and the Linux kernel compilation (`LKC`), to show the effectiveness of KUP. `memtester` is a memory-write intensive application that is used for finding faults in RAM. `memcached` is a in-memory key/value store and we use `memaslap` as a client for load generation. `memaslap` runs on a different machine. `mysql` is a relational database. We use `linkbench` [3], which is a read-dominated benchmark that processes social graphs by using `mysql` as a database. We chose `memcached`, `memtester` and `linkbench` to illustrate memory-heavy workloads, and `LKC` to represent multi-process and computation-heavy program with small working set size (around 25 MB).

Incremental checkpoint iterations. In regard to `FOAM`, KUP's only tunable parameter is the number of iterations used for the incremental checkpoint. To see how the iteration count affects system behavior, we measure throughput, downtime, and amount of data written in the last iteration for `memcached` by varying the iteration count. Even for memory-heavy `memcached`, Figure 3 shows that there were no meaningful differences in both throughput (1-2%) and downtime since the incremental checkpoint quickly converges from the second iteration. This is because, unlike previous studies based on slow network-based incremental checkpointing mechanisms [12, 64], KUP uses fast local storage or memory for checkpointing, making iterations non-critical. For all other experiments in this paper, we set the iteration count to two.

Results. KUP can successfully update the Linux kernel from v4.0 to v4.1 while running the aforementioned programs. Figure 4 shows the exact downtime and the data checkpointed for various schemes of KUP. `PPP` outperforms all other techniques as it only saves the page frame numbers (corresponding to the actual pages' physical address) rather than the actual page contents, thereby decreasing the checkpointed size by orders of magnitude. However, there is not much difference between `PPP` and `FOAM` for `LKC`, because of its smaller working set size.

`FOAM` efficiently supports both incremental checkpointing and on-demand restore. Because of the incremental checkpoint, only the data that gets checkpointed in the



(a) `memtester` **(b)** `memcached` **(c)** `mysql` **(d)** Linux kernel compile

Figure 4: Downtime breakdown of checkpoint-and-restart on various applications when updating a kernel from 4.0 to 4.1 with KUP.

last iteration contributes to the observable downtime. Evidently, due to KUP's simple data structure used for image checkpointing, `FOAM`'s image size remains the same as that of the basic approach. Figure 4 shows negligible downtime for on-demand restore, thus proving the advantage of using on-demand restore for all three approaches: single checkpointed data (marked `Ond`), `FOAM` and `PPP`, against basic restore.

7.3 End-to-end Analysis of KUP's Approach

In this section, we cover an end-to-end performance evaluation of a web service, `memcached` (approximately consuming 5.6 GB with 20% percent write operations), while updating the system's kernel from v4.0 to v4.1. We use `bwm-ng` tool [76] to understand its downtime during update by end-users. Since `memcached` is a memory intensive service caching large amounts of data, where data loss is critical to performance after system update, it fits well for KUP's purpose and effectively shows the impact of our techniques.

Effectiveness of techniques. Figure 5 exhibits the impact of KUP on bandwidth utilization by the client performing `get`/`set` operations on `memcached` server during updates. `PPP` shows the best performance (refer (i)) in terms of shortest downtime and least performance degradation (only 0.68% over the period of 300 seconds) after being restored. Since, `PPP` instantly shifts to `kexec()`, this can be beneficial to applications that are either time critical or that execute for shorter durations. Not only that, `PPP` is storage -independent thanks to its efficient memory mapping storage (see §7.4).

In terms of downtime, `FOAM` performs the same as `PPP` (refer (h) and (i)). This is because `FOAM` provides the best of both worlds by combining both state-of-the-art techniques and has least downtime compared to the individual techniques ((d) and (h)). However, one disadvantage of `FOAM` is that it disrupts the bandwidth during incremental checkpoint (starts around 192 second in (h)), even though it keeps server alive.

The basic technique ((a) and (e)) suffers the most, whereas the incremental checkpoint ((b) and (f)) and on-demand restore ((c) and (g)) techniques perform almost the same, as both reduce their respective downtime in two

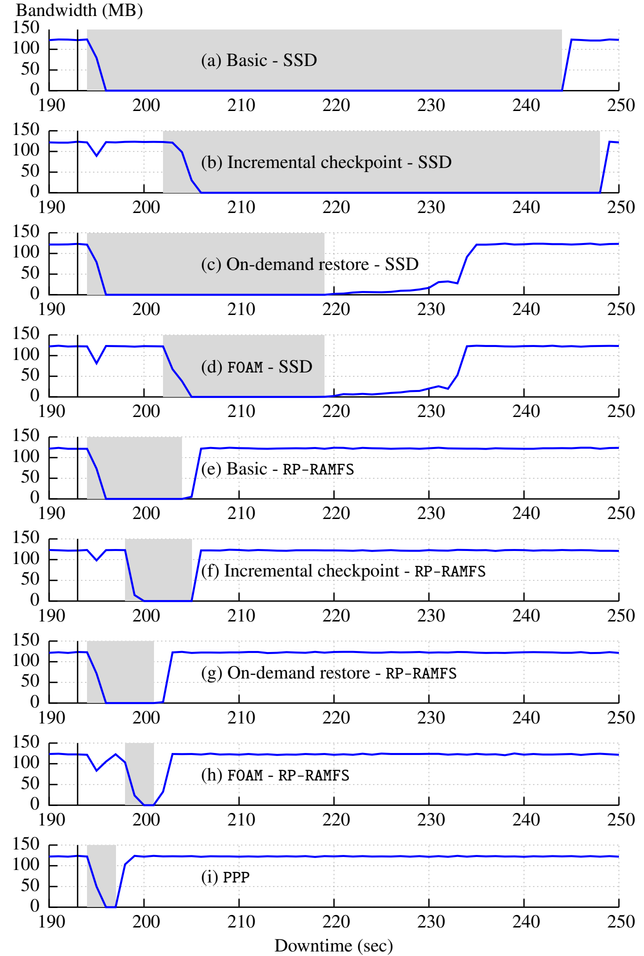


Figure 5: Bandwidth of `memcached` service during a system update. We measured the performance of `memcached` and initiated the kernel update with KUP (a bar near 192 second). From top to bottom, each graph demonstrates properties of each technique, baseline, incremental checkpoint, on-demand restore, and `FOAM` on either SSD or `RP-RAMFS` as a backend medium, and lastly `PPP`. The boxes represent the time from which the network throughput starts decreasing due to the in-progress update.

different steps of OS update. Besides this, the on-demand restore helps the `memcached` server to instantly activate—the bandwidth utilization is non-zero instantly after the OS update (refer (c), (d), (g) and (h)). KUP can further leverage Halite's prefetching approach [81] to improve performance during on-demand restore of `memcached` ((c)

Machine	Soft reboot	kexec() reboot	
	Default	Default	Optimized
1-way 4-core E5-1620/ 64 GB	45.2	2.4	2.4
1-way 8-core E3-1271/ 32 GB	42.9	6.7	6.0
2-way 16-core E5-2630/128 GB	62.9	22.1	9.2
8-way 80-core E7-8870/512 GB	247.5	31.9	25.9

Table 3: Kernel switch time in seconds for different hardware configurations. We used a 1-way 4-core machine for evaluation.

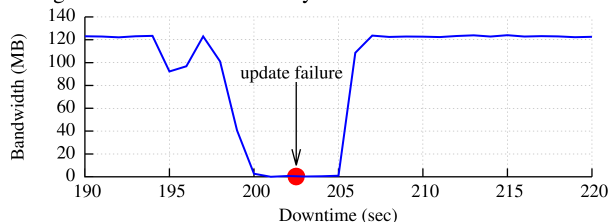


Figure 6: We simulated the update failure by triggering a fault right after system's boot (red mark). KUP's safe fallback mechanism allows memcached to continue running on the previous kernel even after the update failure.

and (d)). Likewise, the incremental checkpoint (refer (b) and (f)) also decreases the downtime during when compared with the basic approach.

Downtime from kernel switch. Although the kernel switch has no effect on application C/R, it adds significant overhead as downtime. Furthermore, this downtime varies with various hardware configurations. Table 3 shows both of the soft and `kexec()` reboot times. The reboot time is the time taken to boot into a new kernel from the previous one. For machines with increasing core count, both of the soft reboot time as well as the default `kexec()` time increase with increasing hardware complexity. On further analysis, we found that most of the time is spent in `purgatory` [31], which runs in between two kernels and initializes hardware components such as PIC and VGA. This holds true even for the bigger machines (>16 cores), which we need to investigate further. In addition, the new kernel also spends time initializing the system services such as network services and filesystem mounting. We reduce the system boot time (refer Table 3) at two places during the new kernel boot: (1) by lazily bringing up each core, which saves six seconds for the 80-core machine, and (2) by skipping the polling of unused PCI slots, thereby saving 8.5 seconds on the 16-core machine.

Safe fallback upon a failure. KUP inherently provides safe fallback mechanism by using `kexec()`. To show its effectiveness, we intentionally inject a fault in the restore process right after the new kernel boots up. As soon as KUP detects a fault, it initiates a downgrade to its previous kernel image, which we stored at the initiation of the update. Figure 6 shows the performance of `memcached` upon such an event. As expected, the downtime of `memcached` becomes approximately two times longer (five seconds) than KUP's update without an error. The safe fallback approach is only used once we are confident we can safely restart the application on the older kernel. We further test the KUP's safe fallback approach with a buggy file system [9] while running Linux kernel compile. Since KUP could not resume compilation in the updated kernel, it reverted to the previous version.

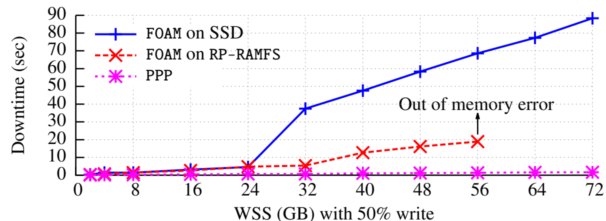


Figure 7: Performance of PPP and FOAM (on RP-RAMFS and SSD) with varying WSS (with 50% write) up to 72 GB, which is larger than a half of system's memory, 128 GB. PPP efficiently supports applications with large WSS, whereas RP-RAMFS fails as it requires free RAM space for checkpointing.

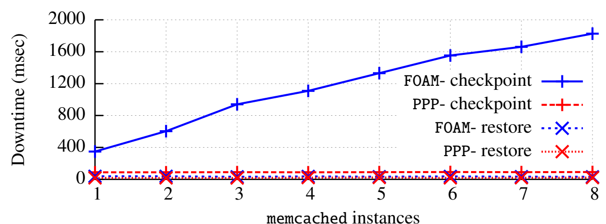
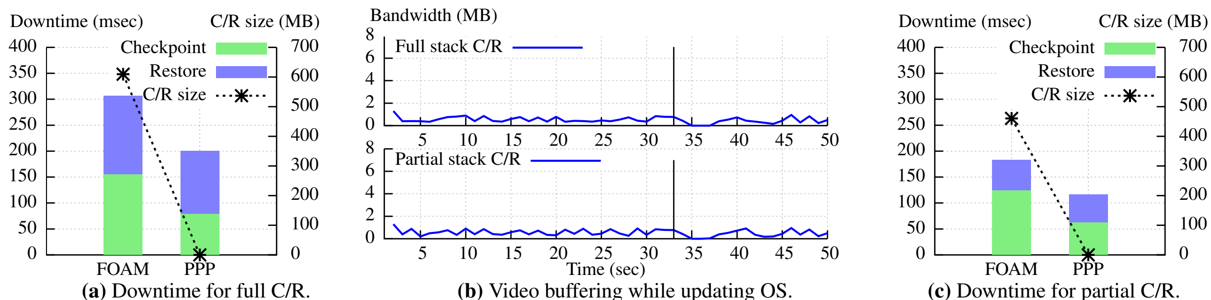


Figure 8: Downtime during checkpoint and restore phase for both FOAM and PPP approaches for multiple instances of `memcached` on SSD.

7.4 Micro-benchmarking FOAM and PPP

In this section, we discuss the impact of PPP and FOAM for various storage media by performing two experiments. This first of these is a microbenchmark-based evaluation which measures the downtime when checkpointing a process with varying working set size (up to 72 GB) with 50% write (Figure 7). The micro-benchmark allocates a certain amount of memory and endlessly dirties the pre-specified part of the allocated memory. By varying the working set size (WSS) from 1 GB to 72 GB on both SSD and RP-RAMFS, we measure the downtime incurred by both techniques. Figure 7 shows the effectiveness of using PPP as it outperforms all the techniques by 9.5-98.3× for SSD and 3.9-14.1× against RP-RAMFS by avoiding redundant copies of process's memory. KUP cannot rely on RP-RAMFS as it fails to work beyond 56 GB and 128 GB is the machine's total memory size.

On the other hand, the second experiment measures the downtime incurred while checkpointing and restarting multiple `memcached` instances (one to eight) on SSD while updating the OS with KUP (Figure 8). The experiment illustrates that (1) KUP is an effective per-application -based C/R mechanism that can checkpoint multiple applications in parallel and (2) PPP is the best approach for updating an OS with multiple applications are run-



(a) Downtime for full C/R. **(b)** Video buffering while updating OS. **(c)** Downtime for partial C/R.

Figure 9: Updating OS with a full-stack media streaming service—`ampache`. (a) shows the downtime observed when performing C/R on `ampache` with KUP. (b) shows the video buffering observed on the client side during OS update. Full web-stack C/R corresponds to updating the complete stack, whereas partial web-stack C/R is the careful C/R performed by the system administrator while updating the OS with KUP.

ning. Figure 8 shows the benefit of using `PPP` against `FOAM`. `FOAM`'s checkpointed data determines its downtime. Thus, with increasing `memcached` instances, KUP's overall downtime increases linearly. However, since `PPP` does not require I/O operations for checkpointing, its downtime remains constant regardless of the number of running instances.

7.5 Supporting the Full Software Stack

Here, we discuss the flexibility of KUP in updating a full software stack made up of typical co-running applications on general purpose servers inside data centers. We use `ampache` as a full software stack. We try to update it with and without the stack's knowledge. `ampache` is a `php` based media streaming service running with `mysql` to store the information about the users and songs. We modify `ampache` to also store movies in `mysql`. Figure 9 shows the downtime incurred because of C/R and as well as the buffering observed on the client side in terms of packets received while streaming a movie clip. On the server side, KUP checkpoints both `apache` and `mysql` and the maximum downtime observed because of C/R is around 300 and 200 milliseconds for `FOAM` and `PPP` respectively (refer Figure 9(a)). On the client side, the buffering stops for around three seconds and then resumes. However, the user does not observe any interruption while watching the video.

Since KUP is a per-application -based OS update mechanism, the administrator can perform partial web-stack C/R during the update by updating only a particular component of the whole service, assuming the administrator has a complete understanding of the full software stack. For example, in our case, KUP should only checkpoint `mysql` and leave the `apache` to the system management daemon. With this approach, the downtime can be further reduced by approximately 100 milliseconds (refer Figure 9(c)). Even though there is no observable improvement on the client side, such selective partial C/R can further reduce downtime in complex software stacks. Moreover, we confirm that the video and song keep on streaming even after the update, which helps us in verifying the correctness of our system.

8 Discussion

In this section, we discuss the limitations of KUP's approaches of using application C/R for instant kernel updates, and we discuss potential optimizations to further reduce the downtime imposed by the kernel switch.

Limitations. KUP inherits the limitations of `criu` for application C/R, such as limited support of `compat` mode of x86-64. This restricts the type of application that KUP can support. Furthermore, application C/R schemes find it difficult to restore the *external resources*. However, in the context of updating OSes, there are many opportunities to overcome these limitations. Because of the shorter time gap between checkpoint and restart in KUP, the host's external interfaces can easily resolve such uncertainty by describing some forms of buffering mechanisms. For example, a `TCP/IP` state will be preserved for about 75 seconds (default in Linux [29]) until the reset packet is fired. As long as the effective downtime is reasonably small, KUP can optimistically restore the checkpointed applications correctly, as in `memcached` §7.3 and full software stack §7.5. Currently, there is an ongoing effort to reliably preserve the TCP/IP states across updates or migrations [16]. Besides this, the current implementation of KUP partially supports unix domain sockets and desktop applications. Another point is that the downtime due to KUP's `FOAM` is still sensitive to the number of cache-to-disk writes. In the future, we would like to explore mechanisms that keep dirty pages across updates.

Using different kernel versions may affect the restoration process, thus adding another limitation to KUP usage. If the new kernel (or old kernel on downgrade) hampers its backward (or forward, respectively) compatibility (e.g., dropping a support of a certain system call, or even changing the interface of certain system calls), the restoration might fail, but we believe this is rare (or unlikely) in practice, as surveyed in §7.1. If the new kernels handle the application-specific states differently, this may result in latent application corruption. However, we are yet

to encounter such unexpected behavior. Another limitation can be that the newer kernel looses the old kernel data. This holds true only for applications that KUP do not checkpoint. Besides that, the kernel's memory state is independent of the checkpointed applications, thereby making them worthless for the checkpoint.

Suitable applications. We believe that KUP is suitable for all types of applications. By using PPP approach, KUP can update any type of application ranging from memory-intensive to I/O-intensive workloads. However, if there is any memory management modification or a page corruption occurs after the update, then the safe fallback technique will also fail. On the other hand, FOAM technique is not a suitable candidate for write-intensive workloads that frequently allocate and free memory pages. However, it provides the guarantee of safely falling back to the previous kernel with more confidence as the process' data is persisted on the disk. Therefore, the system administrators should choose wisely, depending on the criticality of the application and its service.

Optimizing reboot. The major source of downtime in KUP is the in-place kernel switch, depending on `kexec()`; varying from 2.4 sec to 25.9 sec based on the underlying hardware. Although we applied a few basic optimizations to `kexec()` (§6), we can further improve it by adopting a number of promising techniques demonstrated by previous research. For example, Nooks [74] develops shadow drivers techniques that preserve its internal states upon failure, and Live Update [75] uses a similar technique to share its internal state across device driver updates. Another promising direction is to carry over device driver states across reboot as there will be no hardware changes in the middle of system updates.

9 Conclusion

KUP is a simple yet robust update mechanism that instantly updates a running kernel across major kernel versions. KUP checkpoints user applications, then replaces the existing kernel with an updated kernel (any kernel version), and finally restores the checkpointed applications thereafter. KUP achieves its goal of instant kernel update by efficiently merging both checkpoint and restore techniques with the help of an optimized checkpoint format. Moreover, we come up with a complementary memory persistence mechanism that solves the inherent problem of userspace C/R and further improves KUP's performance while catering to disk-less systems. We believe that KUP is the first work that realizes swift kernel updates without modifying any kernel source. This makes KUP robust enough to be used in practice, thus allowing users to enjoy instant system updates for security patches, bug fixes and performance improvements with minimal disruption.

10 Acknowledgment

We thank the anonymous reviewers at ATC'16, and our shepherd, Hani Jamjoom, for their helpful feedback, as well as the operations staff for their proofreading efforts. This research was supported by the NSF award DGE-1500084, ONR under grant N000141512162, DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983].

References

[1] ANDERSON, G., AND RATHKE, J. Dynamic Software Update for Message Passing Programs. In *APLAS* (2012), Lecture Notes in Computer Science, Springer, pp. 207–222.

[2] ANSEL, J., ARYA, K., AND COOPERMAN, G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium* (Rome, Italy, May 2009).

[3] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13, pp. 1185–1196.

[4] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the ACM EuroSys Conference* (Nuremberg, Germany, Mar. 2009), pp. 187–198.

[5] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the 2007 ATC Annual Technical Conference (ATC)* (Santa Clara, CA, June 2007), pp. 26:1–26:14.

[6] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 29–44.

[7] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 ATC Annual Technical Conference (ATC)* (Anaheim, CA, Apr. 2005), pp. 279–291.

[8] BOYD, E. Ext4 corruption associated with shutdown of Ubuntu 12.10, 2012. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1073433.

[9] BRENTAAR. btrfs forced readonly, 2014. https://bbs.archlinux.org/viewtopic.php?id=188900.

[10] CAZABON, C. memtester(8) - Linux man page, 2009. http://linux.die.net/man/8/memtester/.

[11] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)* (2006), pp. 35–44.

[12] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005), pp. 273–286.

[13] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference* (Prague, Czech Republic, Apr. 2013), pp. 211–224.

[14] COMMUNITY. OpenVZ Linux Container, 2005. http://openvz.org.

[15] CORBET, J. Preparing for user-space checkpoint/restore, 2012. https://lwn.net/Articles/478111/.

[16] CORBET, J. TCP connection repair, 2012. https://lwn.net/Articles/495304/.

[17] CUI, W., PEINADO, M., XU, Z., AND CHAN, E. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *Proceedings of the 21st Usenix Security Symposium (Security)* (Bellevue, WA, Aug. 2012), pp. 42–56.

[18] DAVYDOV, V. pram: persistent over-kexec memory file system, 2013. https://lwn.net/Articles/561330/.

[19] DEPOUTOVITCH, A., AND STUMM, M. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the ACM EuroSys Conference* (Paris, France, Apr. 2010), pp. 181–194.

[20] DUMITRAŞ, T., AND NARASIMHAN, P. Why Do Upgrades Fail and What Can We Do About It?: Toward Dependable, Online Upgrades in Enterprise System. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)* (New York, NY, USA, 2009), Springer-Verlag New York, Inc., pp. 18:1–18:20.

[21] DUMITRAS, T., NARASIMHAN, P., AND TILEVICH, E. To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. In *ACM SPLASH Onward* (Oct. 2010), pp. 865–876.

[22] EMELYANOV, P. CRIU: Checkpoint/Restore In Userspace, July 2011. http://criu.org/Main_Page.

[23] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. W. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, CO, Dec. 1995).

[24] FACEBOOK. Open Compute Project, 2015. http://www.opencompute.org/.

[25] FISCHETTI, T. Why is my OS X Yosemite install taking so long?: an analysis, Oct. 2014. http://www.r-bloggers.com/why-is-my-os-x-yosemite-install-taking-so-long-an-analysis.

[26] GHOSHAL, D., RAMKUMAR, S. R., AND CHAUHAN, A. Distributed Speculative Parallelization using Checkpoint Restart. In *ICCS* (2011), Procedia Computer Science, Elsevier, pp. 422–431.

[27] GIUFFRIDA, C., IORGULESCU, C., AND TANENBAUM, A. S. Mutable Checkpoint-restart: Automating Live Update for Generic Server Programs. In *Proceedings of the 15th International Middleware Conference* (2014), Middleware '14, pp. 133–144.

[28] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 279–292.

[29] GODOY, J., HOGBIN, E., KOMARINSKI, M., AND MERRILL, D. TCP keepalive time in Linux, 2015. http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/usingkeepalive.html.

[30] GOEL, A., CHOPRA, B., GEREA, C., MÁTÁNI, D., METZLER, J., UL HAQ, F., AND WIENER, J. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Utah, USA, June 2014), pp. 541–549.

[31] GOYAL, V. kexec: A new system call to allow in kernel loading, 2014. https://lwn.net/Articles/582711.

[32] HALLYN, S., AND GRABER, S. LXC - Linux Containers, 2013. https://linuxcontainers.org.

[33] HARGROVE, P. H., AND DUELL, J. C. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In *Journal of Physics: Conference Series* (2006), p. 494.

[34] HAYDEN, C. M., MAGILL, S., HICKS, M., FOSTER, N., AND FOSTER, J. S. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments* (2012), VSTTE'12, pp. 278–293.

[35] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2012), OOPSLA '12, pp. 249–264.

[36] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *ICDE Workshops* (2011), IEEE.

[37] HEO, T. ptrace: implement PTRACE_SEIZE/INTERRUPT and group stop notification, 2011. https://lwn.net/Articles/441990/.

[38] HINES, M. R., DESHPANDE, U., AND GOPALAN, K. Post-copy Live Migration of Virtual Machines. *ACM SIGOPS Operating System Review*, 3 (July 2009), 14–26.

[39] HOSEK, P., AND CADAR, C. Safe Software Updates via Multi-version Execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)* (2013), pp. 612–621.

[40] HUANG, Y., KINTALA, C., KOLETTIS, N., AND FULTON, N. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on* (June 1995), pp. 381–390.

[41] HYKES, S. Docker: Build, Ship and Run Any App, Anywhere, 2013. https://www.docker.com.

[42] INTERACTIVE, D. Memcached: Distributed memory object caching system, 2003. http://memcached.org/.

[43] JANAKIRAMAN, G. J., SANTOS, J. R., SUBHRAVETI, D., AND TURNER, Y. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2005), 260–269.

[44] KELLY, G. Windows 10 Automatic Updates Start Causing Problems, 2015. http://www.forbes.com/sites/gordonkelly/2015/07/25/windows-10-automatic-update-problems.

[45] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[46] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Recovering from intrusions in distributed systems with Dare. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)* (Seoul, South Korea, July 2012).

[47] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[48] KIM, T., AND ZELDOVICH, N. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 ATC Annual Technical Conference (ATC)* (San Jose, CA, June 2013).

[49] LAADAN, O., AND NIEH, J. Transparent Checkpoint-restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 ATC Annual Technical Conference (ATC)* (Santa Clara, CA, June 2007), pp. 25:1–25:14.

[50] LAWALL, J., AND MULLER, G. Efficient incremental checkpointing of Java programs. In *Proceedings International Conference on Dependable Systems and Networks (DSN)* (2000), pp. 61–70.

[51] LIM, G. Faster Booting in Consumer Electronics. Samsung Electronics, Industry Talks, ATC 2015.

[52] LOWELL, D. E., SAITO, Y., AND SAMBERG, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Boston, MA, Oct. 2004), pp. 211–223.

[53] MAKRIS, K., AND BAZZI, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 ATC Annual Technical Conference (ATC)* (San Diego, CA, June 2009).

[54] NEAMTIU, I., AND HICKS, M. Safe and Timely Updates to Multi-threaded Programs. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 13–24.

[55] NEAMTIU, I., HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (San Francisco, USA, January 2008), pp. 37–49.

[56] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2006), pp. 72–83.

[57] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, Apr. 2013), pp. 385–398.

[58] NOACK, M. Comparative Evaluation of Process Migration Algorithms. 1–51.

[59] NVIDIA drivers not working after upgrade. Why can I only see terminal? http://askubuntu.com/questions/37590/nvidia-drivers-not-working-after-upgrade-why-can-i-only-see-terminal.

[60] ORACLE. Database rolling upgrade using Data Guard SQL Apply. Maximum Availability Architecture White Paper, May 2010.

[61] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), pp. 361–376.

[62] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2014), ISCA '14, pp. 265–276.

[63] PINA, L., AND HICKS, M. Rubah: Efficient, General-purpose Dynamic Software Updating for Java. In *HotSWUp* (2013).

[64] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (Berkeley, CA, USA, 1995), TCON'95, pp. 18–18.

[65] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011), pp. 291–304.

[66] POTTER, S., AND NIEH, J. Reducing Downtime Due to System Maintenance and Upgrades. In *Proceedings of the 19th Conference on Large Installation System Administration Conference (LISA)* (2005), pp. 1–16.

[67] REDHAT. kpatch: dynamic kernel patching, 2014. https://github.com/dynup/kpatch.

[68] SALUSTE, M. Repair or reinstall Windows Update. https://www.winhelp.us/reinstall-windows-update.html.

[69] SINIAVINE, M., AND GOEL, A. Seamless Kernel Updates. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2013), pp. 1–12.

[70] SOULES, C. A. N., SILVA, D. D., AUSLANDER, M., GANGER, G. R., AND OSTROWSKI, M. System Support for Online Reconfiguration. In *Proceedings of the 2003 ATC Annual Technical Conference (ATC)* (San Antonio, TX, June 2003), pp. 141–154.

[71] SPECTOR, L. When Windows Update won't update. http://www.pcworld.com/article/2098429/when-windows-update-wont-update.html.

[72] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, 4 (Aug. 2007).

[73] SUSE. Live Kernel Patching with kGraft, 2014. https://www.suse.com/promo/kgraft.html.

[74] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 4 (Nov. 2004).

[75] SWIFT, M. M., MARTIN-GUILLEREZ, D., BERSHAD, B. N., LEVY, H. M., SWIFT, M. M., MARTIN-GUILLEREZ, D., BERSHAD, B. N., AND LEVY, H. M. Live Update for Device Drivers. Tech. Rep. Technical Report CS-TR-2008-1634, University of Wisconsin Computer Sciences, Mar. 2008.

[76] VGROPP. bwm-ng, 2004. http://linux.die.net/man/1/bwm-ng/.

[77] WHALEY, J. System Checkpointing Using Reflection and Program Analysis. In *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (Kyoto, Japan, Sept. 2001), A. Yonezawa and S. Matsuoka, Eds., LNCS, Springer-Verlag, pp. 44–51.

[78] YAMADA, H., AND KONO, K. Traveling Forward in Time to Newer Operating Systems Using ShadowReboot. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2013), VEE '13, pp. 121–130.

[79] ZARRABI, A., SAMSUDIN, K., AND WAN ADNAN, W. A. Linux Support for Fast Transparent General Purpose Checkpoint/Restart of Multithreaded Processes in Loadable Kernel Module. *Journal Grid Computing*, 2 (2013), 187–210.

[80] ZELLWEGER, G., GERBER, S., KOURTIS, K., AND ROSCOE, T. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, Colorado, Oct. 2014), pp. 17–31.

[81] ZHANG, I., DENNISTON, T., BASKAKOV, Y., AND GARTHWAITE, A. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proceedings of the 2013 ATC Annual Technical Conference (ATC)* (San Jose, CA, June 2013), pp. 1–12.