From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes

Florian Rommel and Christian Dietrich, *Leibniz Universität Hannover;* Birte Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, and Olaf Spinczyk, *Universität Osnabrück;* Daniel Lohmann, *Leibniz Universität Hannover*

https://www.usenix.org/conference/osdi20/presentation/rommel

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes

Florian Rommel [1], *Christian Dietrich* [1], *Birte Friesel* [2], *Marcel Köppen* [2], *Christoph Borchert* [2], *Michael Müller* [2], *Olaf Spinczyk* [2], *and Daniel Lohmann* [1]

[1] Leibniz Universität Hannover [2] Universität Osnabrück

Abstract

Live patching has become a common technique to keep long-running system services secure and up-to-date without causing downtimes during patch application. However, to safely apply a patch, existing live-update methods require the entire process to enter a state of quiescence, which can be highly disruptive for multi-threaded programs: Having to halt all threads (e.g., at a global barrier) for patching not only hampers quality of service, but can also be tremendously difficult to implement correctly without causing deadlocks or other synchronization issues.

In this paper, we present WFPATCH, a wait-free approach to inject code changes into running multi-threaded programs. Instead of having to stop the world before applying a patch, WFPATCH can gradually apply it to each thread individually at a local point of quiescence, while all other threads can make uninterrupted progress.

We have implemented WFPATCH as a kernel service and user-space library for Linux 5.1 and evaluated it with OpenLDAP, Apache, Memcached, Samba, Node.js, and MariaDB on Debian 10 ("buster"). In total, we successfully applied 33 different binary patches into running programs while they were actively servicing requests; 15 patches had a CVE number or were other critical updates. Applying a patch with WFPATCH did not lead to any noticeable increase in request latencies – even under high load – while applying the same patch after reaching global quiescence increases tail latencies by a factor of up to 41× for MariaDB.

1 Introduction

The internet has become a hostile place for always-online systems: Whenever a new vulnerability is disclosed, the respective fixes need to be applied as quickly as possible to prevent the danger of a successful attack. However, it is not viable for all systems to just restart them whenever a patch becomes available, as the update-induced downtimes become too expensive. The prime example for this are operating-system updates, where rebooting can take minutes. However, we increasingly see similar issues with system services at the application level: For example, if we want to update and restart an in-memory database, like SAP HANA or, at smaller scale, an instance of *Memcached* [11] or *Redis* [32], we either have to persist and reload their large volatile state or we will provoke a warm-up phase with decreased performance [26]. With the advent of nonvolatile memory [24], these issues will become even more widespread as process lifetimes increase [19] and eventually even span OS reboots [35]. In general, downtimes pose a threat to the service-level agreement as they provoke request rerouting and increase the long-tail latency.

A possible solution to the update–restart problem is dynamic software updating through live patching, where the patch is directly applied, in binary form, into the address space of the running process. However, live patching can also cause unacceptable service disruptions, as it commonly requires the entire process to become quiescent: Before applying the patch, we have to ensure that a safe state is reached (e.g., no call frame of the patched function f exists on any call stack during patching), which usually involves a global barrier over all threads – with long and potentially unbounded blocking time. In programs with inter-thread dependencies it is, moreover, tremendously difficult to implement such a barrier without risking deadlocks. To circumvent this, some approaches (such as UpStare [22]) also allow patching active functions, which involves expensive state transformation during patch application. Others (like KSplice [3]) probe actively until the system is in a safe state, which, however, is unbounded and may never be reached. Moreover, even in these cases it is necessary to halt all threads during the patch application. DynAMOS [23] and kGraft [29] avoid this at the cost of additional indirection handlers, but are currently restricted to the kernel itself as they rely on supervisor mechanisms. So, while disruption-free OS live patching is already available, live patching of multi-threaded user-space servers with potentially hundreds of threads is still an unsolved problem.

In a Nutshell We present WFPATCH, a wait-free live patching mechanism for multi-threaded programs. The fundamen-

tal difference of WFPATCH is that we do not depend on a safe state of *global quiescence* (which may never be reached) before applying a patch to the whole process, but instead can gradually apply it to each thread at a thread-specific point of *local quiescence*. Thereby, (1) no thread is ever halted, (2) a single hanging thread cannot delay or even prevent patching of all other threads, and (3) the implementation is simplified as quiescence becomes a (composable) property of the individual thread instead of their full orchestration. Technically, we install the patch in the background into an additional *address space (AS)*. This AS remains in the same process and shares all memory except for the regions affected by the patch – which then is applied by switching a thread's AS.

A current limitation of WFPATCH is that we can only patch read-only regions (.text and .rodata). In particular, we cannot apply patches that change the layout of data structures or global variables. However, WFPATCH is intended for hot patching and not for arbitrary software updates and the vast majority of software fixes are .text-only: In our evaluation with OpenLDAP, Apache, Memcached, Samba, Node.js, and MariaDB, this holds for 90 out of 104 patches (87%). For CVE mitigations and other critical issues, it holds for 36 out of 41 patches (88%).

This paper makes the following contributions:

- We analyze the qualitative and quantitative aspects of global quiescence for hot patching and suggest local quiescence as an alternative (Section 2, Section 4).
- We present the WFPATCH wait-free code-injection approach for multi-threaded applications and its implementation for Linux (Section 3).
- We demonstrate and evaluate the applicability of WFPATCH with six multi-threaded server programs (OpenLDAP, Apache, Memcached, Samba, Node.js, and MariaDB), to which we apply patches under heavy load (Section 4).

The patching procedure itself is out of scope for this paper, specifically, how binary patches are generated and what kind of transformations take place when applying them to an AS. Without loss of generality, we used a slightly modified version of Kpatch [30] to generate the binary patches for this paper. However, WFPATCH is mostly transparent in this regard and could be combined with any patch generation framework. We discuss its general applicability, the soundness and limitations and other properties of WFPATCH in Section 5 and related work in Section 6 before we conclude the paper in Section 7.

2 Problem Analysis: Quiescence

Most live-patching methods require the whole system to be in a safe state before the binary patch gets applied. Thereby, situations are avoided where the process still holds a reference to memory that is modified by the update. For example, for a

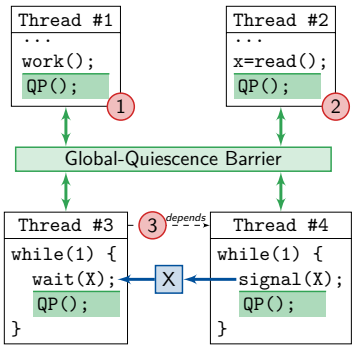

Figure 1: Problems of Global Quiescence. As all threads have to synchronize at the global-quiescence barrier, problems in individual threads can prolong the transition phase: (1) Long-running computations introduce bounded delays, (2) I/O wait leads to (potentially) unbounded barrier-wait times, and (3) inter-thread dependencies force a specific arrival order to avoid deadlocks.

patch that replaces a function f, the system is in a safe state if no call frame for f exists on the execution stack (denoted as *activation safety* in the literature [16]). Otherwise, it could happen that a child of f returns to a now-altered code segment and provokes a crash. While defining and reaching safe states is relatively easy for single-threaded programs, it is much harder for multi-threaded programs, like operating systems or network services.

In general, a safe state of a running process is a predicate Ψ_{proc} over its dynamic state S. For a multi-threaded process, we can decompose this predicate into multiple predicates, one per thread (th1, th2, ...), and the whole process is patchable iff all of its threads are patchable at the same time:

$$\Psi_{\text{proc}}(S) \Leftrightarrow \Psi_{\text{th1}}(S) \wedge \Psi_{\text{th2}}(S) \ldots$$

One possibility to bring a process into the safe state is to use *global quiescence* and insert *quiescence points* into the control flow: When a thread visits a quiescence point its Ψ_{thN} is true and we let the thread block at a barrier to keep the thread in this patchable state. One after another, all threads visit a quiescence point, get blocked at the barrier, and we eventually reach Ψ_{proc} after all threads have arrived. In this stopped world, we can apply all kinds of code patching and object translations [17, 15] as we have a consistent view on the memory.

However, *global quiescence* is problematic as it can take – depending on the system's complexity – a long or even *unbounded* amount of time to reach. Furthermore, eager blocking at quiescence points can result in deadlocks: If the progress of thread A depends on the progress of thread B, thread B must pass by its quiescence points until thread A has reached $\Psi_{\text{A}}(S)$. Even worse, in an arbitrary program, it is possible that $\Psi_{\text{C}}(S)$ and $\Psi_{\text{D}}(S)$ contradict each other such

that $\Psi_{\text{proc}}(S)$ can never be reached. Therefore, programmers need an in-depth understanding of the system to apply global quiescence without introducing deadlocks, and they must take special precautions to ensure that it is reachable eventually.

Figure 1 illustrates these problems. For example, if any thread in the system is performing a long-running computation when the patch request arrives, that is, *Problem 1*, the others will reach the barrier, which is now activated, one by one and stop doing useful work. During this transition-period clients will notice significant delays in response times and requests will queue-up or even time out. We have seen this problem in most of the systems that we examined. For example, Node.js threads perform long-running just-in-time compilation of Javascript code.

Similarly, in *Problem 2*, a thread is waiting on an IO operation. During this potentially unbounded period, other threads will reach the barrier. Again, the overall progress rate deteriorates before it becomes zero during the patching itself. This happens, for instance, when the Apache web server is transferring huge files to a client or executing a long-running PHP script. In an extreme case, the system could even have a thread that is waiting for interactive user input that never comes. Both problems are hard to avoid without changing the complete software structure by the programmer who has to insert quiescence points. Sometimes I/O operations can be quiescence points, but this is application-specific; for example, an I/O operation deep in the call stack or with locks held would be no suitable point for quiescence.

Problem 3 is more subtle and related to inter-thread dependencies. In MariaDB, for instance, worker threads perform database transactions and, thus, have to be synchronized. If a thread that is holding a lock reaches the barrier and blocks, a deadlock will occur if another thread tries to acquire that lock. In this case, the second thread would block and never reach the barrier to free the lock-holding thread. Therefore, a lock-holding thread must not enter the barrier, although its $\Psi_{\text{thN}}(S)$ is true, to avoid the cyclic-wait situation between barrier and lock. More generally speaking, applying global quiescence correctly requires full knowledge about *all* inter-thread dependencies where one thread's progress depends on another thread's progress.

In this paper, we mitigate the aforementioned problems by proposing the concept of local quiescence. Our main contribution is the concept of address-space generations, that is, slightly differing views of an AS that can be assigned on a per-thread basis. This makes it possible to prepare a patch in the background in a new AS and to migrate threads one-by-one to the patched universe. A global barrier is not needed. The approach is "wait-free" in the sense that a thread that has reached a quiescence point ($\Psi_{\text{th}N}(S)$ is true) can be patched immediately. Sections 4.2 and 5 discuss how this approach and its limitations apply to widely-used software projects.

Figure 2 illustrates the difference between the normal "global quiescence" approach (upper half) and the proposed

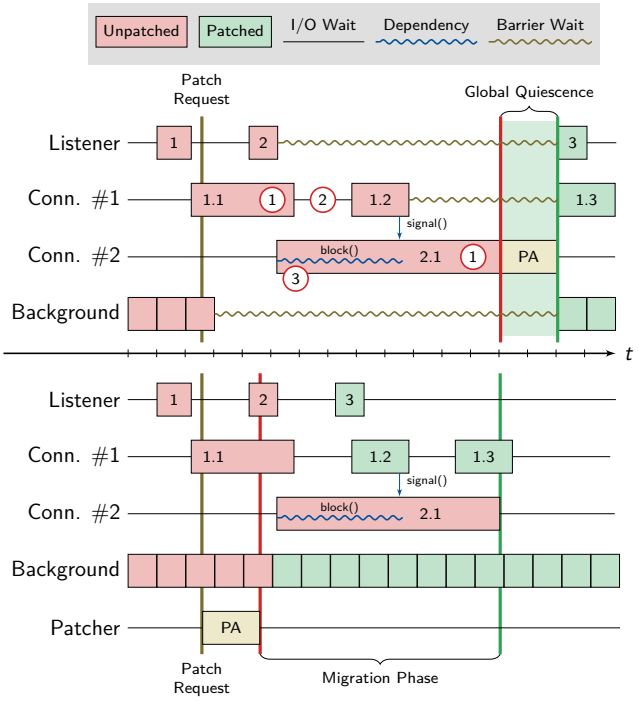

Figure 2: Live Patching a Multi-Threaded Server with Global (upper half) vs. Local (lower half) Quiescence. The global quiescence approach suffers from Problem 1–3 (see Figure 1) while the threads with the local quiescence model can be migrated to the patched state individually.

"local quiescence" (lower half). The scenario is a database server with a "Listener" thread for accepting connections, connection threads ("Conn. #1 and #2") for each client connection, and a "Background" thread for cleanup activities. The patch request comes in asynchronously while the listener is accepting the second connection. At this point in time "Conn. #1" has already started a transaction and is holding a lock. In the upper half (global quiescence) we find all three problems again. For example, the computation time of 1.1 and 2.1 as well as the I/O wait between 1.1 and 1.2 delay the patch application. During this period, the listener does not accept any new connections (request 3) and the background thread is blocked. Furthermore, the programmer must make sure that "Conn. #1" does not block at the barrier before executing 1.2 and releasing the transaction lock, as this would lead the whole system into a deadlock. With local quiescence, each thread can be migrated to the patched program version individually. Thus, no artificial delays are introduced and the quality of service is unaffected. For all but one thread the patch is applied earlier than in the global quiescence case. These seconds might be crucial in the case of an active security attack. Furthermore, deadlocks cannot occur as long as the patched version of the code releases the transaction lock.

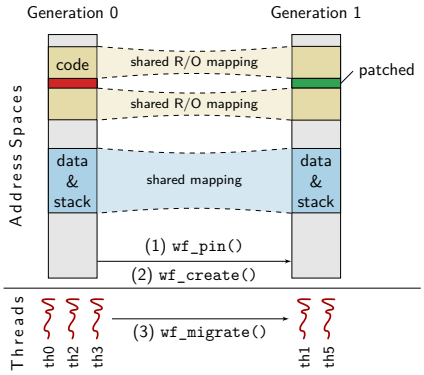

Figure 3: Process during the Wait-Free Patching

3 The WFPATCH Approach

Most previous live-patching mechanisms require a global safe state before applying the changes to the *address space (AS)* of the process. With our approach (see Figure 3), we reverse and weaken this precondition with the help of the OS and a user-space library: Instead of modifying the currently-used AS, we create a (shallow) clone AS inside the same process, apply the modifications there in the background, and migrate one thread at a time to the new AS, whenever they reach a local quiescence point, where their Ψ_{thN} becomes true. In the migration phase, we require no barrier synchronization and all threads make continuous progress. After the migration is complete, we can safely drop the old AS.

While both AS generations exist, we synchronize memory changes efficiently by sharing all unmodified mappings between old AS (Generation 0) and the new AS (Generation 1): We duplicate the *memory-management unit (MMU)* configuration but reference the same physical pages. Thereby, all memory writes are instantaneously visible in both ASs and even atomic instructions work as expected. Only for patch-affected pages, we untie the sharing lazily with existing *copy on write (COW)* mechanisms.

3.1 System Interface

As WFPATCH requires a kernel extension for handling multiple AS generations per process, we introduce four new system calls: `wf_create()`, `wf_delete()`, `wf_pin()`, and `wf_migrate()`. By the integration into the kernel, we are able to modify the AS without halting the whole process.

With `wf_create()`, the kernel instantiates a new AS generation which is a clone of the process's current AS. Any thread, even from a signal handler, can invoke `wf_create()`. AS generations are identified by a numeric ID and can be deleted with the `wf_delete()` system call. We keep AS generations in sync and changes to the AS are equally performed on all generations.

With `wf_pin()`, we can configure, in advance, memory regions that are not shared between AS generations. Within pinned regions, memory writes and page-protection changes will only affect the AS generation of the current thread. Thereby, we are able to have AS generations that differ only in patched pages.

On creation, new AS generations host no threads, but individual threads migrate explicitly by calling `wf_migrate(AS)`. On migration, the kernel modifies the *thread control block (TCB)* to use the patched AS, and the thread continues immediately once the system-call returns. For live patching, threads invoke `wf_migrate()`, via our user-space library, at their local-quiescence points.

3.2 Implementation for Linux

We implemented the WFPATCH kernel extension as a patch with 2000 (added or changed) lines for Linux 5.1. We tested and evaluated WFPATCH on the AMD64 architecture but it should work on every MMU-capable architecture supported by Linux. The basic idea is to clone address spaces in a fork-like manner and rely mostly on the page-sharing mechanism to keep clones lightweight and efficient. In contrast to fork, we do not apply COW, and we synchronize mapping changes between the generations.

The Linux virtual-memory subsystem manages ASs in two layers: The lower layer is hardware dependent and consists of page directories and tables, which have on AMD64 up to 5 (sparsely-populated) indirection levels. On top of this, *virtual memory allocations (VMAs)* group together the non-connected pages into continuous ranges. VMAs contain information for the page-fault handler (e.g. file backing), swapping, and access control. Together, page directories and the list of VMAs, are kept in the *memory map (MM)*, which is attached to a *thread control block (TCB)*.

While Linux normally has a one-to-one relation between MM and process, we discard this convention and let threads in the same process have different MMs, which are *siblings* of each other. Each AS generation has its own distinct MM, which we keep synchronized with its siblings.

Besides adding a list of all existing siblings to the process, we extended each MM to include a reference to a *master MM*. We use this master MM, which is the process's initial MM and its very first generation, to keep track of all shared memory pages. Furthermore, we use the master MM as a fallback for lazily-instantiated page ranges. Therefore, the master persists until the process exits. It cannot be deleted before, even if no thread currently executes in this generation.

When the user calls `wf_pin()` on a memory region, we mark underlying VMAs as non-shared between generations. We allow pinning only on the granularity of whole VMAs and before the first call to `wf_create()`, when the master MM is the only MM in the process.

On `wf_create()`, we duplicate the calling thread's MM

similar to the `fork()` system call when it creates a new child process: For each VMA of the MM, we copy it and its associated page directories to the newly created sibling MM, while all user-pages are physically shared between generations. While `fork()` marks all user pages as COW, we use COW only for pinned VMAs, while most VMAs behave as shared memory regions, which results in the automatic synchronization of user data between generations. By using Linux's COW mechanism for the pinned regions, we are able to lazily duplicate only those physical pages that are actually modified by the patch. After duplication, we select a new generation ID and insert the MM into the process's sibling list.

When a thread calls `wf_migrate()`, we modify its TCB to point to the respective sibling MM. When the thread returns from the system call, it automatically continues its execution in the selected AS generation. Furthermore, each thread that inhabits a generation increases the reference count of the generation by one. Thereby, we ensure that a generation keeps existing as long as threads execute in this address space, even after the user has instructed us to remove the generation (by calling `wf_delete()`). Only after the last thread leaves a deleted generation, we remove the MM and its page directories.

While the system call interface of WFPATCH is straightforward to implement, its integration with other system calls and the page fault handler requires special attention: As some system calls (e.g., `mmap()`, `mprotect()`, or `munmap()`), change a process's AS, we modified these system calls to apply their effects, as long as they touch shared VMAs, not only to the currently active MM but also to all siblings. However, modifying the protection bits for regions in pinned mappings (via `mprotect()`) affects the current MM only.

We also had to modify the page-fault handler, as Linux allows VMAs and the underlying page directory to become out of sync. For example, within a newly-created anonymous VMA, no pages are mapped in the page directory, but they are lazily allocated and mapped by the page-fault handler. By having multiple sibling MMs, we have to make such lazy page loads visible in all generations, when they happen in a shared VMA. We accomplish this by updating not only the current page directory, but also the page directory of the master MM. Upon page faults, we first search the master MM for lazily loaded pages, before allocating a new page.

In order to avoid race conditions between concurrent system calls that modify a process's AS, we use the master MM as a read-write lock that protects all siblings at once. Normally, the MM linked in the TCB is used for this synchronization, but this is insufficient for WFPATCH to synchronize concurrent accesses. Therefore, we decided to use the master MM as a locking proxy and automatically replaced all MM locks with equivalent lock calls to the master MM by using a Coccinelle [27, 28] script. This replacement alone is responsible for 700 of the 2000 lines of changed source code. For processes that do not have multiple generations, this locking strategy imposes no further overhead as the initial MM is the master MM.

In case a process with mutliple AS generations invokes `fork()`, we clone solely the calling thread's currently active generation and make it the only generation in the AS of the child process. This is sufficient, as `fork()` only copies the currently active thread to the newly created process. In order to maintain COW semantics between the forked AS and all generations of the original AS, we have to mark the appropriate page-table entries of all generations as COW pages (i.e. set the read-only flag) – not only the entries of the two directly involved MMs, as we normally would do. This poses a small overhead when forking processes with multiple generations.

When a COW page gets resolved in an AS with multiple generations, we must ensure that the newly copied page replaces the old shared page in all generations, not just in the current one. Therefore, the page fault handler removes the corresponding page-table entry in all generations and maps the new page into the master MM. The master MM fallback mechanism will fill the siblings' page-table entries again (with the copied page) in case of a page fault.

As the AS generations are technically distinct MMs, the migration of a thread to a new AS generation is treated like a context switch between processes. Each generation gets its own *address-space identifier (ASID)* on the processor. Thus, there is no need for a TLB shootdown on AS migrations. Of course, a TLB shootdown (for all generations) is still necessary if access rights become more restricted.

While our kernel extension is a robust prototype, several features are still missing (e.g., userfaultfd, a mechanism to handle page faults in user space) and some are not extensively tested (e.g., swapping, NUMA memory migration, memory compaction). However, for none of these features, we see any fundamental problem that would conflict with our approach or cause a significant deterioration in the performance of the overall system after adding full support.

3.3 User-Space Library

Our proposed system interface (see Section 3.1) allows a process to create new AS generations, to migrate individual threads, and to delete old generations. In order to utilize this system-call interface for live patching with local quiescence, we built a user-space library around this system-call interface. In the following, we will describe its API as well as its usage in a multi-threaded server with one thread per connection (see Figure 4).

At start, the user initializes and configures our library with `wf_init()`: With `track_threads`, she promises to signal the birth and death of threads such that our library can keep track of all currently active threads and delete old AS generations after the last thread has migrated away. Alternatively, the user can configure a callback that returns the current number

```
int main(void) {
 wf_config_t config = {
  .track_threads = 1,
  .on_migration_start=&f,
 };
 wf_init(config);
 wf_thread_birth();
 signal(RTMIN, sigpatch);
 ...
 while (true) {
   int c = accept();
   spawn_worker(c);
   wf_quiescence();
 }
}
```

```
void worker(int fd) {
  wf_thread_birth();
  while (!done) {
    x = read(fd);
    work(x);
    wf_quiescence();
  }
  wf_thread_death();
}
```

```
void sigpatch(int) {
  char *p;
  p = find_patch();
  wf_load_patch(p);
}
```

Figure 4: Usage of our User-Space Library

of threads. Furthermore, the user can install other callbacks that we invoke at certain points of the migration cycle. In the example, we invoke `f()` when the new AS is ready for migration and, thereby, give the user the possibility to trigger blocked threads in order to speed up the migration phase. With the initialization, the library starts the patcher thread, which pins the text segment, creates new AS generations, and orchestrates the migration phase.

As initiation of live updates and the location of patch files is application specific, we leave this to the user application and only provide a library interface to start the patching application (`wf_load_patch()`). This function instructs the patcher thread to load a binary patch from the file system and apply it in a new AS generation. In our current implementation, `wf_load_patch()` supports ELF-format patches created by Kpatch [30]. These patches are loaded, relocated, and all contained functions are installed in the cloned text segment via unconditional jumps at the original symbol addresses. Furthermore, all references within the patch to unmodified functions, global variables, and shared-library functions are resolved dynamically. Afterwards, the patcher marks the new AS as ready for migration and sleeps until all thread have migrated.

At the thread-local quiescence points, the user has to call `wf_quiescence()` periodically, which checks if a new AS generation is available and ready for migration. If so, the library calls `wf_migrate()` in the context of the current thread and increases the number of migrated threads. After all threads have migrated, the patcher thread is woken, deletes the old AS generation and ends the migration phase.

4 Evaluation

We evaluate WFPATCH with six production-quality infrastructure services on a Linux 5.1 kernel running the Debian 10 Linux distribution (codename "buster", released on 2019-07-10). Table 1 provides a brief overview of the respective Debian packages for OpenLDAP, Apache HTTPD, Memcached, Samba, Node.js, and MariaDB. We use the initial Debian 10 packages and prepare the server executables for dynamic patching with global and local quiescence (Section 4.1). Our goal is to apply all patches published by the Debian maintainers until 2020-05-09 for these binaries with our approach (Section 4.2). This situation mimics a system administrator who maintains a long-running server running one of these services.

For quantitative evaluation, we measure and compare the service latency while applying a binary patch with global and local quiescence (Section 4.3), respectively, as well as the memory and run-time overheads caused by WFPATCH (Section 4.4).

4.1 Implementation of Quiescence

As outlined in Section 2, implementing global quiescence in a complex multi-threaded program can be a difficult undertaking causing three problems in general: Long-running computations (Problem 1) and waiting for I/O (Problem 2) prolong the transition period, which results in deteriorating service quality, while inter-thread dependencies necessitate stopping the threads in an application-specific order to avoid deadlocks (Problem 3). In the following, we describe how we encountered these three problems in our evaluation targets and how they manifest in their structure and fundamental design decisions. Besides the steps we had to take in order to achieve global quiescence, we also describe how we can reach local quiescence for each of the projects we evaluated.

OpenLDAP The OpenLDAP server (`slapd`) uses a listener thread that accepts new connections and dispatches requests as work packages to a thread pool of variable, but limited size ($\leq$ 16 threads). Each work package is processed by a single worker thread, which alternates between computation and blocking I/O until the request is answered.

For global quiescence, we submit a special task to the thread pool. The executing worker pauses all other workers with the built-in pause-pool API, which can only be called from a worker context, and visits a quiescence point on behalf all worker threads. Since the listener thread waits indefinitely for new connections, we need to introduce an artificial timeout (1 second) to provoke quiescence points periodically. For *local* quiescence, we only introduce a quiescence point before the listener waits for a new connection and after a worker thread completes a task.

As worker threads execute client requests as a single task without visiting a quiescence point, complex requests (problem 1), slow client connections (problem 2), and large result sets (problem 2) prolong the barrier-wait time.

Apache The default configuration of the Apache web server (`httpd`) uses the built-in multi-processing module `event`, which implements one dedicated listener thread and a configurable number of worker threads (default: 25). That listener thread handles all new connections, all idle network sockets, and all network sockets whose write buffers are full to

avoid blocking of the worker threads. In its main loop, the listener thread periodically checks for activity on the listening, idle, and full network sockets by using the Linux system call `epoll()` with a timeout of up to 30 seconds, which can cause Problem 2. Once a network socket becomes active, the listener thread unblocks the next free worker thread to serve that socket.

We introduce one quiescence point into each main loop of the listener and worker threads. For global quiescence, however, we have to make sure that the listener thread enters global quiescence after all worker threads have done. Otherwise, some worker threads may block indefinitely because the listener thread cannot unblock them anymore (Problem 3). When returning from global quiescence, the listener's timeout queue needs to be fixed manually to account for the elapsed time spent in global quiescence.

Implementing *local* quiescence in Apache is straightforward by just introducing the same quiescence points without bothering about deadlocks nor timeouts.

Memcached Memcached is event-driven and uses 10 threads in the default configuration: Four worker threads wait for network requests and the completion of asynchronous I/O tasks. One listener thread accepts new connections and wakes up at least every second to update a timestamp variable. Both the workers and the listener use `libevent` to orchestrate event processing. Furthermore, three background threads wait on a condition variable, while two other threads use `sleep()` to wake up periodically with a maximal period of one second.

For global quiescence, we use a built-in notify mechanism to wake up the all workers immediately, even if they are blocking in `libevent`. For the listener thread, we have to use `event_base_loopbreak()` to interrupt the event-processing loop. Unfortunately, this only sets a flag that the listener checks within the aforementioned one-second period. Furthermore, we have to signal the three condition variables to wake up the associated maintenance threads, as they would block indefinitely otherwise. The two sleeping threads will, eventually, reach the quiescence point, but waking them is not necessary to avoid deadlocks. For *local* quiescence, we use the same quiescence points and the same wake-up strategy as for global quiescence.

While the main operation of Memcached is event-driven and, therefore, the threads do not block on I/O operations, the periodic maintenance threads and the listener thread provoke barrier-wait times of up to one second (Problem 2).

Samba For live patching, Samba's `smbd` was especially challenging as it uses a combination of process-based and thread-based parallelization. For each connection, which can live for hours and days if established by a client mount, the process is forked and uses internally a thread pool to parallelize requests. This thread pool shrinks and grows dynamically with the request load, while idling worker threads retire only after a given timeout (1 second). Technically, these workers wait on a condition variable with a one-second timeout and are woken when a listener thread enqueues a received request. In order to issue a patch request, the system administrator has to inform all processes to initiate the patching process.

For global quiescence, we have to signal each worker's condition variable. A woken worker checks whether the barrier is active and visits a quiescence point instead of retiring early as an idle worker. For local quiescence, we just inserted quiescence points after the condition wait and after a received network request.

As each request is limited in size, `smbd` only suffers from problem 2 when workers wait for a send operation to complete. However, as the thread pool dynamically grows to up to 100 threads under heavy load, the overall barrier-wait peaks when the server is most intensely used.

Node.js For asynchronous I/O operations, Node.js spawns one thread that executes a `libuv` loop. For computation, Node.js uses one work queue for immediate tasks executed by a variable number (n) of worker threads, and a second queue for delayed tasks, which is serviced by a dedicated thread. Each worker executes tasks sequentially and offloads I/O to the `libuv` thread.

For binary patching, we introduce quiescence points in the I/O thread and after a worker completes a task. For global quiescence, we submit n empty tasks to the immediate work queue and one task to the delayed work queue. For the `libuv` thread, we had to manually signal a semaphore to prevent deadlocks (problem 3). For *local* quiescence, we only submit one task to the delayed work queue and use the same quiescence points otherwise.

As all computation, including the just-in-time compilation, is dispatched via work queues, a long job (problem 1) will increase the barrier-wait time even though the Javascript execution model is inherently event-driven.

MariaDB MariaDB's `mysqld` supports two thread models: one thread per connection, which is the default, or a pool of worker threads. In both cases, a separate listener thread accepts new connections and passes them to connection or worker threads, and a total of 30 helper threads handle offloaded I/O and housekeeping. We implemented patching support for both thread models.

Judging from its public bug tracker, SQL query evaluation appears to be MariaDB's most error-prone component. We therefore limit the global barrier to threads parsing or executing SQL statements and do not add quiescence points to listener or helper threads. Even so, our global quiescence implementation faces all the three challenges outlined in Section 2.

Slow queries, such as complex SELECT or large INSERT statements, increase the barrier-wait time as threads perform the computation (problem 1) without visiting a quiescence point. Depending on the query and the size of the database, this can lead to excessive wait times.

In both threading variants, idle threads are cached in anticipation of new work before being retired. In one thread

per connection mode, the hard-coded timeout is five minutes; for the thread pool, it defaults to one minute (problem 2). As barrier-wait times of over a minute are unrealistic for any global-quiescence integration, we utilize preexisting functions to wake up all cached threads for patching. We introduce a new global patch variable to distinguish between a wake up due to a new connection, server shutdown, or patching in one thread per connection mode.

MariaDB supports *SQL transactions*, which are an atomic group of SQL statements whose effects are only visible to other connections after the transaction has completed. As MariaDB serializes transactions which access the same data via locks, threads encounter request- and database-induced dependencies (problem 3). If a thread reaches the barrier while holding a transaction lock, other threads that try to get this lock before their next visit at a quiescence point will deadlock. In one thread per connection mode, we handle this by skipping the barrier if the connection holds a transaction lock. For the thread pool, this does not suffice: as each thread handles several connections, waiting on the barrier is forbidden as long as any open transaction is present.

For *local* quiescence, visiting a quiescence point is possible regardless of the transaction state. Apart from that, we use the same quiescence points and wake-up strategies as for global quiescence.

Global vs. Local Quiescence Summarized, we encountered Problem 1 in three projects (OpenLDAP, Node.js, MariaDB), Problem 2 in four projects (OpenLDAP, Memcached, Samba, MariaDB), and Problem 3 in four projects (OpenLDAP, Apache, Node.js, and MariaDB). While Problem 1 and 2 in combination with global quiescence only affect service quality, Problem 3 forced us to introduce different application-specific dead-lock avoidance techniques into our benchmarks. Thereby, we repeatedly experienced set-backs and spurious deadlocks while navigating the often complex web of existing inter-thread dependencies – achieving global quiescence was the hardest part of our evaluation! In contrast, incorporating WFPATCH was straightforward as we only had to identify the local-quiescence points before patch application could start.

4.2 Binary Patch Generation

To demonstrate the applicability of live patching in running user-space programs, we created a set of binary patches for the aforementioned six network services (see Table 1). For each project, we use the current version that is shipped with Debian 10.0 as a baseline against which we apply patches. In Debian, it is common to select one version of a project for a specific Debian release and have the maintainer backport critical patches onto that version.

For five projects (except MariaDB), we systematically inspected the Debian source package for maintainer-prepared patches that touch the source code of the network service. Debian patches reflect critical updates that an expert on the service selected for this specific version. Therefore, we consider these patches as a good candidate set for live patches that a system administrator wants to apply. We also review the subset of patches with a CVE entry to get statistics of highly-critical security updates.

For MariaDB, the source package contains no patches: Debian follows MariaDB releases instead of backporting individual patches. Therefore, we processed all commits in the 10.3 branch of the MariaDB repository, starting with the 10.3.15 release shipped with Debian 10.0. Each set of commits that references a single bug tracker entry classified as *Bug* with a severity of at least *Major* related to `mysqld` is a source patch. As the bug tracker does not reference CVE numbers, we use patches with a severity of at least *Critical* instead.

From these source-code patches, we manually select those which only influence the .text segment and do not alter data structures or global variables, as such patches are currently out of scope for our mechanism. In Table 1, we see that most patches that are hand-selected by a maintainer are text-only patches; for CVE patches, the correlation is even higher. For MariaDB, where we have a large set of critical patches, 91 percent of the patches exclusively modify the program logic. We therefore conclude that a mechanism which supports live patching with a restriction to code-only changes is nevertheless a useful contribution for keeping running services up to date.

As patch generation, in contrast to patch application, is not among our intended contributions, we use the Kpatch toolchain, which was developed for live-updating the Linux kernel, to prepare binary patches from source code changes. Unfortunately, due to shortcomings in Kpatch, we could not create binary patches for all text-only changes. Especially MariaDB and Node.js, which are implemented in C++, show a low success rate. In the lower half of Table 1 we summarize, over all generated binary patches, the average number of changed object files, modified function bodies, and the size of each patch text segment.

We verified our mechanism by applying each patch into the corresponding service while processing requests. We successfully applied all binary patches generated by Kpatch with our user-space library using thread migration at local quiescence points.

In total, we successfully applied 33 different binary patches including 15 CVE-relevant patches. For OpenLDAP, Apache, and Samba, we were able to apply all generated patches sequentially into the running process. This was not possible for MariaDB because the patches are not applicable to a common base version due to the amount of patches that we could not generate with Kpatch. Making the patches applicable sequentially in MariaDB would have meant to backport them to the initial version, like the Dabian maintainers did for the other projects.

		OpenLDAP (slapd)	Apache (httpd)	Memcached	Samba (smbd)	MariaDB* (mysqld)	Node.js
Release		2.4.47	2.4.38	1.5.6	4.9.5	10.3.15	10.19.0
All Patches (CVE)	[#]	13 (2)	10 (10)	1 (1)	2 (2)	74 (26)	4 (0)
.text Only (CVE)	[#]	9 (2)	7 (7)	1 (1)	2 (2)	67 (24)	4 (0)
kpatch'able (CVE)	[#]	9 (2)	7 (7)	1 (1)	2 (2)	16 (5)	0 (0)
⌀Mod. Files	[#]	1.11	1.71	1	1	1.19	–
⌀Mod. Functions	[#]	3.67	13.71	1	5.5	2.94	–
⌀Patch Size	[KiB]	13.02	56.94	43.91	9.23	15	–

* For MariaDB, no Debian patches were available and MariaDB maintainers do not relate bugs to CVEs. We instead took patches with severity $\geq$ *Major* from the project's bug tracker as base; numbers in brackets denote patches with severity $\geq$ *Critical*.

Table 1: Evaluation projects and patches (of which CVE-related) since Debian 10.0 release

4.3 Request Latencies

In order to quantify the service quality benefits of local quiescence and incremental thread migration over the barrier method, we perform an end-to-end test for our selected projects. For each project, we define a benchmark scenario and measure the end-to-end request latencies encountered on the client side, while we (a) generate new AS generations and migrate threads, or (b) stop all threads at a global barrier. For this, we extended our user-space library to also support global-quiescence states via the barrier method. We periodically send patch requests to the same process and skip the actual text-segment modification in these tests, while still inducing barrier-wait times on the one side and AS-creation overheads on the other side. Thereby, we achieve a high coverage of different program states at patch-request time, while keeping the comparison fair.

All experiments are conducted on a two machine setup. The server process runs on a 48-core (96 hardware threads) Intel Xeon Gold 6252 machine clocked at 2.10 GHz with 374 GiB of main memory. The clients execute on a 4-core Intel Core i5-6400 machine running at 2.70 GHz with 32 GiB of main memory. Both machines are connected by a Gigabit link in a local-area network.

On the server side, we start the service, wait 3 seconds for the clients to come up and then trigger a local-quiescence migration or global-quiescence barrier sync every 1.5 seconds. By this patch-request spreading, the impact of the barrier method can cool down before the next cycle starts. On the client side, we measure the end-to-end latency of each request. In total, we simulate at least 1000 patch requests for each benchmark.

For OpenLDAP, 200 parallel client connections send LDAP searches that result in 50 user profiles from a database with 1000 records. For Apache, we use ApacheBench to download a 4 MiB sample file 50,000 times using 10 parallel connections; due to the shared Gigabit link, a download takes about 350 ms when no threads are blocked on the global quiescence barrier. For Memcached, 50 client connections request a random key from a pool of 1000 cached objects of 64 KiB. For MariaDB, which we operate in the one-thread-per-connection mode, four sysbench `oltp_read_only` connections continuously perform transactions with five simple SELECT statements, while four background connections – whose latency we do not monitor – execute transactions with 2000 statements. For Node.js, we developed an example web service that encodes a request parameter in a QR-code, wraps it in a PDF, and sends the resulting "ticket" back to the client. We use the `wrk` tool to simulate 10 parallel clients that repeatedly request a new ticket. For Samba, we mount the exported file system on the client machine (`mount.cifs`) and use the sysbench `fileio` benchmark with 32 threads, a block size of 16 KiB, and an R/W ratio of 1.5 to measure file I/O latencies.

Please be aware that these scenarios are chosen as examples to demonstrate the possible impact of barrier synchronization. Resulting latencies are highly dependent on the workload and can be smaller, but also vastly larger in other scenarios. For example, by executing long-running SQL queries on MariaDB or downloading large files from an Apache server, the barrier-wait times, and therefore the latency of the global-quiescence method, can be increased arbitrarily.

Figure 5 shows latency histograms (with logarithmic y axis) for local and global quiescence, as well as the 99.5 response-time percentile. In all benchmarks, we see a significant increase in tail latency which ranges from a factor of 0.97× (Node.js) to 41× for MariaDB. While the results for OpenLDAP, MariaDB, and Samba directly show the latency impact of a global barrier, the other results require explanations. For Memcached, three out of ten threads perform one-second waits, resulting in latencies of up to one second. For Apache, local quiescence shows a narrow latency distribution with the predicted peak at 350 ms while global quiescence shows a broadened distribution. This is due to the benchmark's network-bound nature: the last worker to reach the barrier enjoys the unshared 1 Gigabit link to finish its last request, while all requests arriving after the patch request are impacted by the barrier-wait time. In Node.js, the percentiles

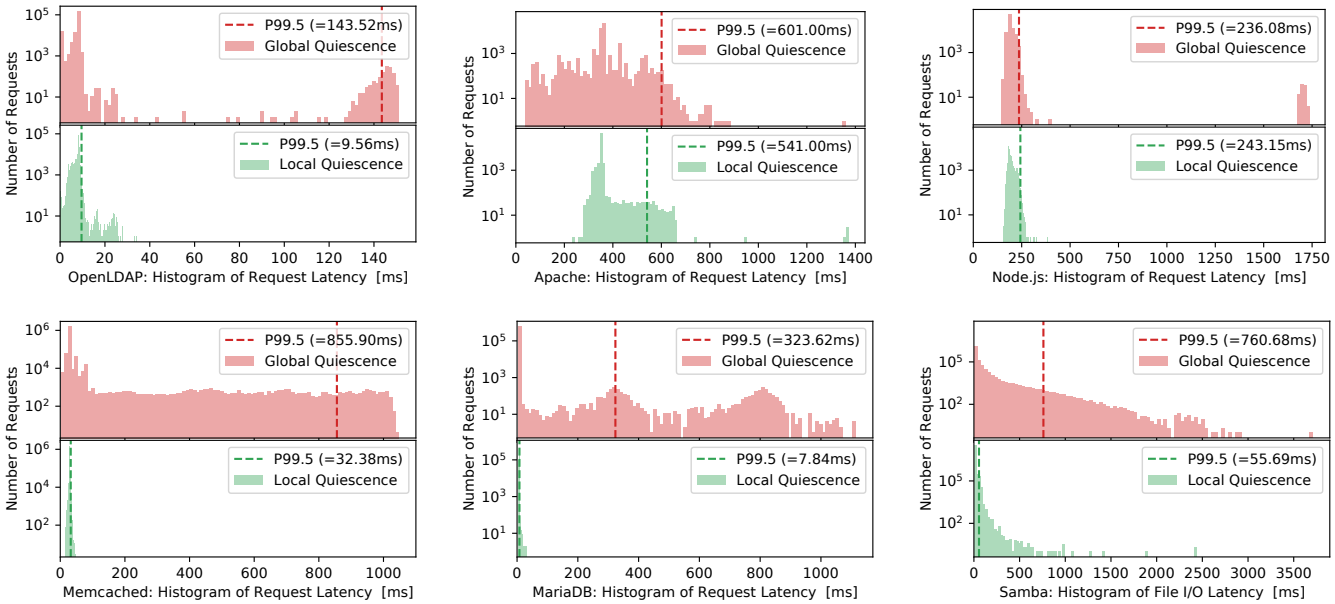

Figure 5: Request Latencies during Live Patching

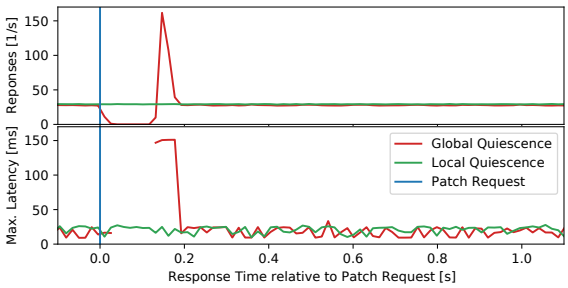

Figure 6: OpenLDAP Response Rates during Quiescence

are almost equal as the longest encountered barrier-wait time (18 ms) is still shorter than the average request duration's jitter (193±53 ms). However, we observe individual barrier-wait times of more than 1.5 seconds.

For a deeper understanding of the encountered service quality directly after a patch request, we analyze OpenLDAP responses during 1000 patch requests. We correlate each received response to the previous patch request and plot them according to their relative receive time; zero being the patch request. Figure 6 shows response rate and maximum observed latency. After a patch request, the response rate in the global-quiescence case rapidly decreases, while the latency stays at its normal value. After the workers reach the barrier, no responses are recorded until the listener has reached the barrier. After global quiescence is reached, `slapd` ramps up again and processes the request backlog built up in the meantime. This causes the response rate to spike, but those responses are so late that we see a significant latency increase before the service returns to normal operation. With WFPATCH, no impact, neither on the response rate nor on the maximum latency, can be observed.

4.4 Memory and Run-Time Overheads

For each patch application, our kernel extension duplicates the MMU configuration, creates a new AS generation, and performs one AS switch per thread in order to migrate it to the new generation. To quantify the impact of these operations, we measure the MMU configuration size and perform run-time micro benchmarks of AS creation and switching times for each server application. We run the benchmarks under load (see Section 4.3) to provoke disturbance and lock contention in the kernel.

We measure the memory overhead caused by duplicate MMU configurations by sequentially applying as many patches as possible. In Table 2, we report the difference in MMU configuration size before and after the patch application. As the other data-structure additions required for our extension are negligible in size, this is the total memory overhead during patch application. Due to the non-deletable master MM (see Section 3.2), this overhead becomes permanent for patched processes: starting with the first additional generation, we carry the load of this additional MM. We do not introduce a memory overhead for processes which do not use AS generations.

For the run-time overhead, we perform two micro benchmarks. (1) The patcher thread creates a new AS generation and immediately destroys it. (2) The patcher thread migrates back and forth between two AS generations (2 switches). We

	Memory [KiB]	Runtime Penalty Create [µs]	Switch [µs]
OpenLDAP	412	298±47	7±7
Apache	680	429±17	7±6
Memcached	132	88±23	7±6
MariaDB	516	1339±38	7±6
Node.js	1808	2171±139	8±7
Samba	256	672±54	5±5

Table 2: Address Space Management Overhead

	Upstream [µs]	WFPATCH [µs]
(a) Anonymous Mapping	0.40±0.12	0.42±0.15
(b) File Mapping	0.50±0.14	0.50±0.15
(c) Read Fault	0.87±0.18	0.87±0.20
(d) Write Fault	1.23±0.29	1.25±0.32
(e) COW Fault	1.79±0.35	1.81±0.39

Table 3: Steady-State Run-Time Overhead

execute each scenario a million times in a tight loop and report, in Table 2, the average operation time alongside its standard deviation. We see that the creation and destruction of AS generations scales with the size of the process's virtual address space. Only for Samba and MariaDB, the creation overhead is, compared to the MM size, disproportionately higher than for the other four benchmarks. This is caused by a higher number of file-backed VMAs in Samba and MariaDB that take longer to duplicate. The `wf_migrate()` call is a constant-time operation.

In the implementation of our approach, we tried to minimize overhead for applications that do not use WFPATCH. Memory consumption overhead is limited to few additional fields in the thread control block (2 pointers + 2 integer fields), the memory map (3 pointers), and the structure that represents a memory mapping (1 boolean field). In terms of run-time overhead, WFPATCH adds code in two critical places in the kernel: the mapping modification functions and the page-fault handler. In order to assess the run-time impacts, we performed micro benchmarks on our modified kernel and on an upstream kernel with the same version and configuration. To evaluate mapping modifications, we map and unmap either (a) an anonymous memory region or (b) a file mapping and measure the time of the `mmap()` system call. The results do not show a significant difference between the kernels (see Table 3). For page faults, we issue (c) a read operation or (d) a write operation on a previously untouched portion of an anonymous mapping. To also capture (e) copy-on-write resolution, we write to a page that is also mapped by a forked process. Each of the five measurements was repeated 10 million times.

5 Discussion

Benefits of Local Quiescence The main benefit of patching threads individually is the simplified establishment of quiescence and the avoidance of a global barrier that causes a deterioration in performance. Thereby, WFPATCH provides latency hiding for Problem 1 and 2 (Section 4.1) and mitigates Problem 3.

Nevertheless, in the light of the rare event of applying a live patch to an application, the overhead and tail latency of global quiescence may seem negligible. However, the benchmarks presented in Section 4.3 do not necessarily represent a real-world or worst-case scenario: We use a single client machine with a fast, stable, and reliable local network connection to the server. Furthermore, we aimed for a controlled and uniformly distributed load pattern for the sake of reproducibility and in order to fairly compare the relative impact of global vs. local quiescence. In a real-world scenario, connection latencies will vary wildly or may be even under control by an active attacker. As barrier-synchronized global quiescence couples the progress of all threads in the system, it is much more prone to such latency variations – the latency impact is dictated by the slowest (in case of an attacker: stalling) thread to reach the barrier. With WFPATCH and local quiescence, all other threads will not only continue working, but also have the patch applied immediately. Even if a thread stalls forever, the only damage is an AS generation that will never get freed, while the patched server continues to answer requests.

Lightweight AS Generation For the lightweight AS generation, our current implementation copies the whole MM in `wf_create()`, including VMAs and the page directories. This leads to the differing memory and creation overheads that we observed for our benchmark scenarios (Table 2).

While we consider these overheads as reasonable for the purpose, they could nevertheless be reduced further if we implement the different generations to share parts of their page-directory structure. This is possible for shared VMAs, as the underlying page tables always reference the same physical pages. In fact, we currently even pay for not sharing them by extra efforts to keep page tables synchronized among AS generations via the master MM. However, VMAs cover page ranges with arbitrary start/end index, while the page-directory tree covers page rages on a power-of-two basis, so implementing such sharing is not trivial. To the best of our knowledge, Linux itself does not employ page-table sharing between address spaces, even though this would probably be beneficial for the implementation of the fork system call.

Code Complexity The current implementation of WFPATCH adds a certain amount of complexity to the kernel (see Section 3.2). This stems from its interaction with the already-complex kernel memory-management subsystem. One reason is that Linux targets numerous different architectures and exploits most of their individual capabilities. Secondly, the

kernel itself provides many features and often chooses performance over simplicity (e.g., fine granular page table locking or code duplication in the mapping functions). Apart from that, WFPATCH's complexity is also caused by the tight connection between address spaces and processes in Linux. As the idea of AS generations itself is straightforward, the complexity of our kernel extension could be reduced significantly if we decoupled the two concepts of address spaces and processes in general. That would not only serve our approach, but may even promote other ideas and development [21, 9], such as the decoupling between threads and processes did.

Applicability The general applicability of WFPATCH is potentially limited by (a) the restriction to .text/.rodata-patches only and (b) the preparation of the respective target program. With respect to (a) this depends on the intended use case: We consider WFPATCH currently as an approach to apply hot fixes to a server process under heavy load – in order to prolong the time it needs to be restarted until the next maintenance window. For this use case, our results show that the vast majority of patches (87%) are .text-only and, therefore, applicable; for critical patches (CVE mitigations) this number is even higher (88%). Regarding (b), the WFPATCH user-space library simplifies the preparation of the target program to support hot patching, but like in other approaches that support multi-threaded applications, it is up to the developer to identify and model the respective safe points to apply a patch. With WFPATCH, however, it becomes significantly easier to find these points as they need to be only locally quiescent. In our evaluation, the hardest part of integrating WFPATCH into the six multi-threaded server programs was the global barrier we needed solely for the comparison between local and global quiescence.

Soundness and Completeness Proving the soundness of a dynamic update is an undecidable problem [14], even though type checking and static analysis can help to mitigate the situation in some cases [1]. With WFPATCH, we have the additional complexity of *incomplete patches*, that is, some threads still execute the old code, while others already use the patched version. This, however, imposes additional correctness issues only if the code change actually influences inter-thread data/control dependencies, such as the implementation of a producer–consumer protocol. In practice, this is a rare situation – none of the analyzed 90 .text-only patches fell into this category. Nevertheless, a possible solution in such cases would be to gradually give up the wait-free property by implementing *group quiescence* among the dependent threads, while all other threads can still migrate wait-free at their local quiescence point. Compared to global quiescence, group quiescence would still be less debilitating for overall response time and easier to implement in a deadlock-free manner.

In general, if some thread has not yet passed its point of local quiescence, it is either blocking somewhere in an I/O or still actively processing a request that arrived before the patch was triggered. In both cases, it is at most this one request that may still be processed using the old version. This would also be the case with global quiescence – only that with global quiescence based on barriers all other threads have to wait (see Figure 6); if global quiescence is determined by probing for a safe state (such as in Ksplice [3]), the other threads continue processing requests using the unpatched version. If the respective thread hangs forever, global quiescence based on barriers would result in a deadlock, while with probing the patch would never get applied. With WFPATCH, the patch will be applied as far as possible: All new requests will be processed with the new code – a server may even be patched while under an active DDOS attack. Technically, an incomplete patch means that the process will stay in two (or even more) ASs forever.

Overall, local and global quiescence make a different tradeoff between correctness requirements and ease of patch applicability: While applying patches with global quiescence requires less upfront thought about the correctness of a patch as it provokes no transition period, it may be hard or even impossible to introduce the patch in the system. On the other hand, although it is harder to show that a patch is suitable for local-quiescence patching, finding local-quiescence points is easier and patch application has only minimal impact on the system's operation. We believe that many time-critical updates (e.g., additional security checks) have such a localized impact on the code that the guarantees of local-quiescence patching are sufficient for a large number of changes.

Generalizability For the sake of simplicity, we chose to adapt the Kpatch binary-patch creation for our evaluation and implemented a loader for such patches for user-space programs (Section 3.3). Thereby, we also inherit the limitations of Kpatch regarding granularity and installation of patches: Patches work at the granularity of functions; they are installed by placing a jump at the original symbol address to redirect the control flow to the patched version. This bears some overhead, but is arguably the most widespread technique to apply run-time patches [1, 23, 3, 29, 30, 5, 6]. Furthermore, only quiescent (inactive) functions can be patched. While this limitation is a lot less problematic with WFPATCH due to the fact that quiescence is reduced to local quiescence (inactive in the currently examined thread), it nevertheless prevents patching of top-level functions.

It is important to note, though, that these are restrictions of the employed patching mechanism, not of its wait-free application offered by WFPATCH, which is the main contribution of this work. Integration with more sophisticated patching methods [17, 15, 22] could mitigate these limitations while keeping the WFPATCH benefits. For instance, UpStare can patch active functions by an advanced *stack reconstruction* technique [22]. Hence, it does not require quiescence, but nevertheless has to halt the whole process for patch application and reconstruction of all stacks. In conjunction with WFPATCH, this expensive undertaking could be performed in the background while other threads continue to make progress.

Data Patching While our toolchain already supports the introduction of new data structures and global variables, we currently do not support patches that change existing data-structures or the interpretation of data objects. Such patches are generally difficult [36] as a transform function that migrates the system state to the new representation must be applied to all modified objects in existence. Current live-patching systems rely on the developer to supply these transform functions [17, 15], while language-oriented methods for semi-automated transformer generation exists [20, 18, 25].

With local quiescence, state transfer becomes more difficult as two threads that touch the same data can execute in different patching states. Therefore, an extension to data patches would require bidirectional transform functions that are able to migrate program state back and forth as needed. MMU-based object migration on read and write accesses via page faults can be used to trigger the migration of individual objects between AS generations. Similar mechanisms are used to provide virtual shared memory on message-passing architectures [2]. However, for thread-local state only a unidirectional transform function is required.

Other Applications In a nutshell, WFPATCH provides means for run-time binary modifications in the background, which can then be applied wait-free to individual threads. Besides run-time binary patching, the fundamental mechanism could be useful for many further usage scenarios.

For example, every just-in-time (JIT) compiler has to integrate newer, more optimized versions of functions into the call hierarchy while the program is executing. With WFPATCH, the JIT could prepare complex changes and rearrangements across multiple functions in the background in a new AS generation and then apply them, without stopping user threads, by migrating the benefiting threads incrementally to the updated AS. Furthermore, as our kernel extension supports an arbitrary number of AS generations, the JIT could provide specialized thread-local function variants with the same start address, keeping all function pointers valid.

In a similar manner, an OS kernel could transparently apply path-specific kernel modifications [31] on a per-thread basis. For example, the kernel could use a different IRQ subsystem that is only used if a thread with real-time priority gets interrupted.

AS generations can not only be used to provide a differing code views between threads, but also data views. This can be employed to provide isolation for security and safety purposes. For example, a server application could make encryption keys only be present in a special AS generation; the other generations would have an empty mapping in this place. Even individual threads could live in their own AS generations in order to keep sensible data private but share all the other mappings with their sibling threads. The major benefit compared to using `fork()` with distinct processes is that all mappings are shared by default and modifications to the mapping are implicitly synchronized – the address spaces do not diverge. Moreover, threads can easily switch back and forth between generations. Litton et al. [21] made a similar suggestion in form of thread-level address spaces, which, however are not synchronized, thus being similar to `fork()` in this respect.

In general, WFPATCH is able to provide classical cross-cutting concerns (debugging, tracing, logging) with a thread-local view of the text segment. For example, a debugger may limit the effect of trace- and breakpoints to the actually debugged threads or use the unoptimized program only during the debugging session. Also, the user could enable tracing, logging, assertions, or behavioral sanitizers (e.g., Clang's UBSan) for individual threads.

6 Related Work

Dynamic patching of OS kernels has a long history in research [13, 4, 5, 12] and is now actually used in production systems [3, 29, 30]. In contrast, the suggested frameworks to patch user-level processes [20, 25, 6, 22, 17, 15, 12] are still not broadly employed.

The DAS [13] operating system incorporated an early run-time updating solution on module-level granularity. It requires absolute quiescence of a module to be patched, realized by locks. K42 [4] exploits its strict object-oriented design to enable live kernel updates. The event-driven nature with short-lived and non-blocking threads makes it relatively easy to define a safe state for concurrent patching.

The Proteos [12] microkernel provides built-in means for process-level live updates based on automatic state transfer. Like our wait-free patching technique, they employ MMU-based address spaces, but unlike our approach the goal is not a seamless thread-by-thread migration. Instead, the process is halted during the update procedure, while the separate address space provides for an easy rollback.

Most live-patching frameworks work on function-level granularity [1, 23, 3, 29, 30, 5, 6], which can be considered as a natural scope for changes while still providing for relatively fine-grained updates. A patched version of the function is loaded and installed via placing a trampoline jump at the beginning of the old function body (function indirection). Barrier blocking is the classical way to reach global quiescence to safely apply the trampoline. Ksplice [3] avoids this by polling for global quiescence instead: The whole kernel is repeatedly stopped and checked for a safe state before the function indirection gets installed. While this avoids a global barrier, all threads have nevertheless to be halted for the check and to apply the patch. Furthermore, probing is an unbounded operation, so the patch may be applied late or never.

DynAMOS [23] and kGraft [29] also avoid global barriers by extending the function indirection method: By (atomically) placing additional redirection handlers between the trampoline and the jump target, they can decide on a per-call basis which version of a function (original/updated) should be used. This has some similarities to our address-space migration

technique as in both methods the patched and the unpatched universe coexist while the transition is in progress; however, in contrast to our approach, the redirection method induces a performance penalty in this phase. Atomicity is reached by rerouting the call through debug breakpoints during the patch process; on SMP systems this furthermore requires IPIs to all other cores to flush instruction caches. This approach is limited to patching on function granularity and has only been explored for kernel-level patching, whereas WFPATCH targets user-level processes and allows for arbitrary large (or small) in-place binary modifications, which in principle also includes changes to (read-only) data.

LUCOS [5] tries to solve this by requiring the to-be-patched kernel to run inside a modified XEN hypervisor, which is able to atomically install trampoline calls by halting the VM. The virtualization layer is also used to enable page-granularity state synchronization between the different versions of a function. POLUS [6] brings this idea to user space and relies on the underlying operating system (ptrace, signals and mprotect) instead of a hypervisor. Again, all threads are halted while the trampoline gets installed.

Ginseng [25] makes use of source-to-source compilation in order to prepare C programs for dynamic updating. It inserts indirection jumps for every function call and every data access, but does not support multi-threaded programs. Function indirections are also used by many other language-oriented dynamic-variability methods, such as dynamic aspect weaving [7, 10, 34] or function multiverses [33], which, however, do not address quiescence in multi-threaded environments.

Ekiden [17] and Kitsune [15] provide dynamic updates by replacing the whole executable code and transferring all program state at dedicated *update points*, which constitute points of global quiescence implemented by barriers in the case of multi-threading. UpStare [22] goes one step further by allowing run-time updates at arbitrary program states, enabled by its *stack reconstruction* technique. However, updating multi-threaded programs is also based on halting all threads. The authors even suggest inserting the respective checks in long-lived loops and to avoid blocking I/O.

Duan et al. present a comprehensive solution for patching vulnerable mobile applications on the binary level [8]. However, patching takes place when the program starts and not during later run time.

The idea of decoupling address spaces and processes has also been described before: El Hajj et al. [9] provide freely switchable address spaces in order to enlarge virtual memory and to support persistent long-lived pointers. However, they do not target live patching and their address spaces are intended to be decoupled from each other, whereas WFPATCH provides extra means to synchronize most regions among address space generations.

Litton et al. [21] allow for multiple "light-weight execution contexts" (lwC) per process and the possibility for threads to switch between them. After creation, where the file-descriptor table and the AS are copied (like fork), lwCs are decoupled entities and can diverge significantly from each other. In contrast, our AS generations offer a gradually differing view of the same AS without decoupling other parts of the execution context (i.e. file-descriptor tables). Thereby, all threads retain a synchronized view of process state, which is necessary for incremental thread migration.

7 Conclusion

WFPATCH provides a wait-free approach to apply live code patches to multi-threaded processes without "stopping the world." The fundamental principle of WFPATCH is that a code change is not applied to the whole process at once, which requires a state of global quiescence to be reached by all threads simultaneously, but incrementally to each thread individually at a thread-specific state of local quiescence. Hence, (1) no thread is ever halted, (2) a single hanging thread cannot delay or even prevent patching of all other threads, and (3) the implementation gets easier as quiescence becomes a (composable) local property. The incremental migration is provided by means of multiple generations of the virtual address space within the updated process. After preparation of an updated address space, threads switch generations at their local quiescence points, while they are still able to communicate with threads in other generations via shared memory mappings.

We implemented WFPATCH as a Linux 5.1 kernel extension and a user-space library, and evaluated our approach with six major network services, including MariaDB, Apache and Memcached. While live patching at points of global quiescence with a barrier increases the tail-latency of client requests by up to a factor of 41×, we could not observe any disruption in service quality when live patches were applied wait-free with WFPATCH. In total, we successfully applied 33 different binary patches into running programs while they were actively servicing requests; 15 patches had a CVE number or were other critical updates.

WFPATCH brings us closer to an ideal live patching solution for multi-threaded applications by solving the response-time issue with a latency hiding patch-application mechanism. This opens further research opportunities on advanced patching techniques.

Acknowledgments

We thank our anonymous reviewers and our shepherd Andrew Baumann for their constructive feedback and the efforts they made to improve this paper. We also thank Lennart Glauer for his work on an early WFPATCH prototype.

This work was supported by the German Research Council (DFG) under the grants LO 1719/3, LO 1719/4, SP 968/9-2.

The source code of WFPATCH *and the evaluation artifacts are available at:*
https://www.sra.uni-hannover.de/p/wfpatch

References

[1] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. Opus: Online patches and updates for security. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM '05, USENIX Association, pp. 19–19.

[2] APPEL, A. W., AND LI, K. Virtual memory primitives for user programs. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (1991), pp. 96–107.

[3] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: automatic rebootless kernel updates. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys '09)* (New York, NY, USA, Mar. 2009), J. Wilkes, R. Isaacs, and W. Schröder-Preikschat, Eds., ACM Press, pp. 187–198.

[4] BAUMANN, A., HEISER, G., APPAVOO, J., SILVA, D. D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), pp. 279–291.

[5] 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* (New York, NY, USA, 2006), VEE '06, ACM, pp. 35–44.

[6] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 271–281.

[7] DOUENCE, R., FRITZ, T., LORIANT, N., MENAUD, J. M., DEVILLECHAISE, M. S., AND SUEDHOLT, M. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)* (Chicago, Illinois, Mar. 2005), P. Tarr, Ed., ACM Press, pp. 27–38.

[8] DUAN, R., BIJLANI, A., JI, Y., ALRAWI, O., XIONG, Y., IKE, M., SALTAFORMAGGIO, B., AND LEE, W. Automating patching of vulnerable open-source software versions in application binaries. In *2019 Network and Distributed System Security Symposium (NDSS 2019)* (2019).

[9] EL HAJJ, I., MERRITT, A., ZELLWEGER, G., MILOJICIC, D., ACHERMANN, R., FARABOSCHI, P., HWU, W.-M., ROSCOE, T., AND SCHWAN, K. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, Association for Computing Machinery, p. 353–368.

[10] ENGEL, M., AND FREISLEBEN, B. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)* (Chicago, Illinois, Mar. 2005), P. Tarr, Ed., ACM Press, pp. 51–62.

[11] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5–.

[12] 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 '13)* (New York, NY, USA, 2013), ACM Press, pp. 279–292.

[13] GOULLON, H., ISLE, R., AND LÖHR, K.-P. Dynamic restructuring in an experimental operating system. *IEEE Transactions on Software Engineering SE-4*, 4 (1978), 298–307.

[14] GUPTA, D., JALOTE, P., AND BARUA, G. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering 22*, 2 (1996), 120–131.

[15] HAYDEN, C. M., SAUR, K., SMITH, E. K., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Trans. Program. Lang. Syst. 36*, 4 (Oct. 2014), 13:1–13:38.

[16] HAYDEN, C. M., SMITH, E. K., HARDISTY, E. A., HICKS, M., AND FOSTER, J. S. Evaluating dynamic software update safety using systematic testing. *IEEE Transactions on Software Engineering 38*, 6 (2012), 1340–1354.

[17] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *2011 IEEE 27th International Conference on Data Engineering Workshops* (Apr. 2011), pp. 179–184.

[18] HICKS, M., MOORE, J. T., AND NETTLES, S. Dynamic software updating. *SIGPLAN Not. 36*, 5 (May 2001), 13–23.

[19] HSU, T. C.-H., BRÜGNER, H., ROY, I., KEETON, K., AND EUGSTER, P. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the 12th*

European Conference on Computer Systems (EuroSys '17) (2017), ACM, pp. 468–482.

[20] LEE, I. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin-Madison, 1983.

[21] LITTON, J., VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 49–64.

[22] MAKRIS, K., AND BAZZI, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX '09, USENIX Association, pp. 31–31.

[23] MAKRIS, K., AND RYU, K. D. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)* (New York, NY, USA, Mar. 2007), T. Gross and P. Ferreira, Eds., ACM Press, pp. 327–340.

[24] MEENA, J. S., SZE, S. M., CHAND, U., AND TSENG, T.-Y. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters 9*, 1 (2014), 526.

[25] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 72–83.

[26] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[27] PADIOLEAU, Y., LAWALL, J. L., MULLER, G., AND HANSEN, R. R. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)* (New York, NY, USA, Mar. 2008), ACM Press.

[28] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J. L., AND MULLER, G. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)* (New York, NY, USA, 2011), ACM Press, pp. 305–318.

[29] PAVLÍK, V. kgraft: Live patching of the linux kernel, 2014. https://www.suse.com/media/presentation/kGraft.pdf, visited 2019-08-05.

[30] POIMBOEUF, J., AND JENNINGS, S. Introducing kpatch: Dynamic kernel patching, 2014. https://rhelblog.redhat.com/2014/02/26/kpatch, visited 2019-08-05.

[31] PU, C., MASSALIN, H., AND IOANNIDIS, J. The Synthesis kernel. *Computing Systems 1*, 1 (1988), 11–32.

[32] REDISLAB. Redis, 2019. http://redis.io, visited 2019-07-21.

[33] ROMMEL, F., DIETRICH, C., RODIN, M., AND LOHMANN, D. Multiverse: Compiler-assisted management of dynamic variability in low-level system software. In *Fourteenth EuroSys Conference 2019 (EuroSys '19)* (New York, NY, USA, 2019), ACM Press.

[34] SCHRÖDER-PREIKSCHAT, W., LOHMANN, D., GILANI, W., SCHELER, F., AND SPINCZYK, O. Static and dynamic weaving in system software with AspectC++. In *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Track 9* (2006), Y. Coady, J. Gray, and R. Klefstad, Eds., IEEE Computer Society Press.

[35] SELTZER, M., MARATHE, V., AND BYAN, S. An NVM carol: Visions of nvm past, present, and future. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 15–23.

[36] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis mutandis: Safe and predictable dynamic software updating. In *ACM SIGPLAN Notices* (01 2005), vol. 40, pp. 183–194.