Wait-Free Code Patching of Multi-Threaded Processes

Florian Rommel, Lennart Glauer, Christian Dietrich, Daniel Lohmann
{rommel, glauer, dietrich, lohmann}@sra.uni-hannover.de
Leibniz Universität Hannover, Germany

Abstract

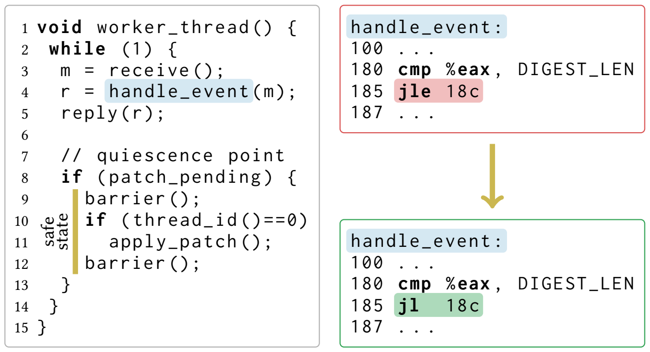
In the operation and maintenance phase of a deployed software component, security and bug-fix updates are regular events. However, for many high-availability services, costly restarts are no acceptable option as the induced downtimes lead to a degradation of the service quality. One solution to this problem are live updates, where we inject the desired software patches directly into the volatile memory of a currently running process. However, before the actual patch gets applied, most live-update methods use a stop-the-world approach to bring the process into a safe state; an operation that is highly disruptive for the execution of multi-threaded programs.

In this paper, we present a wait-free approach to inject code changes into a running multi-threaded process. We avoid the disruption of a global barrier synchronization over all threads by first preparing a patched clone of the process's address space. Into the updated address space, we gradually migrate individual threads at predefined quiescence points while all other threads make uninterrupted progress. In a first case study with a simple network service, we could completely eliminate the impact of applying a live update on the request latency.

1 Introduction

In the usual paradigm of software development, we frequently modify code, recompile and restart a test instance of a program, and verify that our changes implement the desired behavior. With deploying the software, this rate of changes drops significantly as the project enters the operation and maintenance phase. However, due to bug fixes and security updates, the change rate of actively maintained software never really drops to zero. Therefore, we must not only deliver a static update file, but we also have to apply these changes to our deployed systems.

PLOS'19, October 27, 2019, Huntsville, ON, Canada
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7017-2/19/10...$15.00
https://doi.org/10.1145/3365137.3365404



```
1  void worker_thread() {
2   while (1) {
3    m = receive();
4    r = handle_event(m);
5    reply(r);
6
7    // quiescence point
8    if (patch_pending) {
9      barrier();
10     if (thread_id()==0)
11       apply_patch();
12     barrier();
13   }
14  }
15 }
```

```
handle_event:
100 ...
180 cmp %eax, DIGEST_LEN
185 jle 18c
187 ...
```

```
handle_event:
100 ...
180 cmp %eax, DIGEST_LEN
185 jl  18c
187 ...
```

Figure 1. Multi-Threaded Network Server with Bug Fix. This program is manually prepared to apply live updates at run time. At the quiescence point (line 7ff), a barrier synchronization is used to bring the process into a safe state before the bugfix patch (at address 185) gets applied (line 11).

However, not for all systems it is viable to just restart a running instance, since the update-induced downtimes become too expensive. The prime example for this are operating-system updates as rebooting requires us to stop processes, reinitialize the hardware, and restart all applications. While this problem was long confined to the OS domain, we increasingly see similar issues on the application level: For example, if we want to update and restart an in-memory database, like *memcached* [6] or *Redis* [21], we either have to persist and reload its large volatile state or we will provoke a warm-up phase with decreased performance [18]. For other high availability systems, even if they are stateless, downtimes pose a threat to the service-level agreement as they provoke request rerouting and increase the long-tail latency. With the advent of nonvolatile memory [16], these issues will become even more widespread as process lifetimes increase [12] and eventually even span over several OS reboots [23].

One solution to the update–restart problem is dynamic software updating through live patching, where the update is directly applied, in binary form, to the code and data segment of the running process. For the OS itself, this possibility has become widely used in practice, while solutions for applications are still not commonly employed.

Quiescence Points in Multi-Threaded Programs 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 patch that replaces a function f, the system is in a safe state if no call frame for f exists on the execution stack. 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, this predicate can normally be decomposed 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 define a safe state is to insert *quiescence points* into the source code: At these points, a thread is in a patchable state and its Ψ_{thN} becomes true. In order to reach Ψ_{proc}, these points are part of a barrier synchronization that block threads at the quiescence point until all threads have arrived. In this stopped world, we can apply all kinds of code patching and object translations [10, 11] as we have a consistent view on the memory.

Figure 1 shows a condensed version of a network service whose code is prepared to apply a live patch with the barrier method: The process executes several worker threads (left), which receive and handle messages in an infinite loop. In between messages, the thread comes to the quiescence point, checks if a patch is pending (as shown on the right), and synchronizes at the barrier. After all threads have arrived, thread 0 applies the bug-fix patch (making it take effect for all threads), before all threads resume message handling.

However, this live-patching method comes at the cost of an operation disruption for multi-threaded programs: With the blockade of more and more threads, the overall progress rate deteriorates, before it becomes zero during the patching itself.

It is this time period, between the initiation of the patching process and its completion, that we want to make wait-free. We achieve this by migrating threads incrementally between two address spaces; the unpatched and patched one. Both address spaces remain in the same process and share all memory except for the modified regions. In particular, we claim the following contributions:

- We propose the idea of wait-free code patching on the basis of address-space generations and incremental thread migration.
- We provide a proof-of-concept implementation for Linux that provides code-segment updates and safe state definition based on quiescence points.
- We demonstrate the applicability and the benefits of our approach with two case studies.

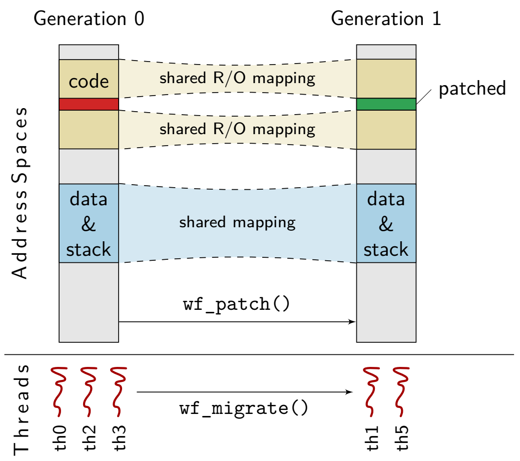


Figure 2. Process during the Wait-Free Patching. For the patching, we clone the original address space (Gen. 0) such that all mappings are shared. In the new address space (Gen. 1), we un-share the patched pages and modify them. Successively, threads migrate from the original to the patched generation with an explicit system call (`wf_migrate()`).

2 The Wait-Free Patching Approach

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 2), we reverse and weaken this precondition with the help of the OS: Instead of modifying the currently-used AS, we create a (shallow) clone AS inside the same proccess, apply the modifications there, and migrate one thread at a time to the new AS, whenever their Ψ_{thN} becomes true. In the migration phase, we require no barrier synchronization and all threads make continuous progress. After all threads have migrated, we can safely drop the old AS.

While both AS generations exist, we must synchronize changes to memory between them. We achieve this efficiently by sharing all mappings from the old AS (Generation 0) with the new AS (Generation 1): We duplicate the *memory-management unit (MMU)* configuration (page directories and page tables) 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 those pages that are affected by the live update, we untie the shared mapping, copy the old contents to a new physical page, and apply the patch.

In this paper, we exclusively focus on updates to read-only memory, like the code segment or constant data. Thereby, we exclude situations where a thread in the old generation wants to make an update to memory that has been modified by the patch. Furthermore, we also exclude updates that modify the data layout or the meaning of already allocated objects. Thereby, all writable memory regions have the same interpretation before and after the update. We also demand that the developers annotate quiescence points, which are valid for all considered updates, in the program.

2.1 System Interface

As our wait-free method is provided as an OS service, we introduce two new system calls: `wf_patch()` and `wf_migrate()`. With the integration into the kernel are we able to modify the AS without bringing the whole process to a halt.

With `wf_patch()`, the application starts the update process and prepares the new AS. Any thread, even from a signal handler, can invoke `wf_patch()` with an array of changes: Each change consists of (1) a pointer to the patch data, (2) the length of the patch data, and (3) the virtual address where the patch should be applied. With this interface, we only support the application of in-place updates. If an updated function body is larger than the original function, or if memory should be moved, the user of our interface must provide an appropriate change list.

Within the OS, `wf_patch()` duplicates the AS of the current process, which results in a new virtual address space that references the same physical user pages. For all pages that are affected by the change list, we copy its old contents into a newly allocated page and apply the patch. We update the new AS, which is not yet inhabited by any thread, to reference the updated page.

While `wf_patch()` only prepares the patched AS, `wf_migrate()` performs the actual incremental thread migration. The thread invokes this system call explicitly at a thread-local quiescence point. The OS modifies the thread control block to use the patched AS and with the system-call return, the thread directly continues.

Besides memory writes, we also have to keep both ASs synchronized when the process requests a change to its virtual-memory mappings. This includes adding and removal of mappings (i.e., `mmap()`, `munmap()`) and the change of page protections (i.e., `mprotect()`). While we explicitly forbid modifications to the mapping that would affect patched areas, we allow other changes and apply those to both ASs.

In our current implementation, we do not yet support the creation of new threads or child processes (i.e., `fork()`) during the transition. However, in principle, new threads should inherit the AS of its parent thread. For `fork()`, the new process only contains a clone of the invoking thread, which should be placed in the same AS generation.

2.2 Implementation for Linux

We implemented our wait-free approach prototypical for Linux 5.1 and the AMD64 architecture. On this architecture, the virtual memory mappings of an AS are encoded as page directories of up to 5 (sparsely-populated) indirection levels. Thereby, all nodes in this page-directory tree are page sized (4K) and can hold up to 512 sub nodes.

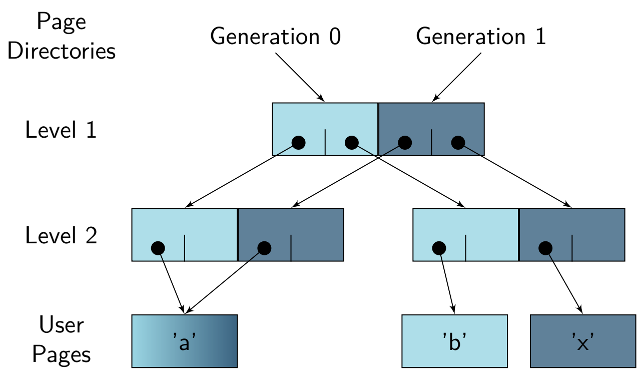


Figure 3. Page-Directory Duplication. Instead of keeping two independent page-directory trees, we always allocate two 8K pages for each directory. In this example, each directory level can hold up to two children and a live update (`'b'`$\rightarrow$`'x'`) is currently in progress.

In order to provide multiple ASs per process, we decided to always allocate two page directories (see Figure 3) and keep them synchronized, even when no migration is in progress. For this, we modified the page-directory allocation to return two consecutive 8K aligned pages instead of one 4K page. With this *page-directory duplication*, we can switch between generations by flipping a single bit in a pointer. This prevents us from having to maintain an explicit mapping for each directory.[1] While this technique allows for fast synchronized modification of both directories, it doubles the memory requirement for the MMU configuration.

In order to integrate our method with the rest of the Linux memory subsystem, we modified the normal setter functions for page directories. These setters, which are often called with a pointer to a page-directory sublevel, are used to manifest all high-level AS changes (i.e., `mmap()`) in the actual MMU configuration. Instead of modifying only the given page directory, the setters now also flip the generation bit and perform the same modifications for the sibling directory. For `wf_patch()`, we use specialized setters that only affect one generation.

For the bookkeeping of the AS generations, we have extended the thread control block (`task_struct`): For each thread, we store the current AS generation and the target generation. While the former is used during the context switch to load the correct page directory, the latter is used for `wf_migrate()`. Furthermore, we store the number of threads that are not yet migrated to the target generation.

Another technical issue during the context switch are *address-space identifiers (ASIDs)*, which are used to distinguish between *translation-lookaside buffer (TLB)* entries from

[1]This technique is similar to KPTI (previously KAISER [9]), which switches between separate kernel and user page directories. Our implementation is compatible with KPTI.

different ASs and help to avoid full TLB flushes. Therefore, we cannot use the same ASID for both generations, although both ASs are largely equal. Again, we solve the problem by duplicate allocation: Instead of one core-local ASID, we allocate two ASIDs when a thread gets dispatched for the first time on a processor and decide upon its current AS generation which ASID to use. On AMD64, this is no limitation as 4095 ASIDs are supported and Linux uses at most six of them ASIDs – which we increased to twelve.

3 Case Study: Updating a Network Service

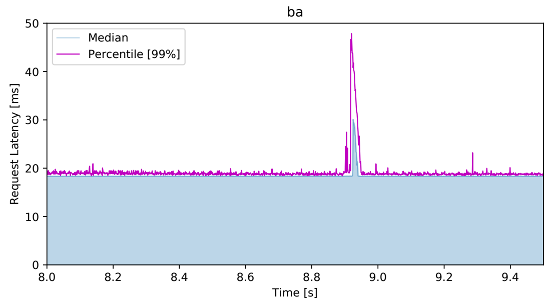
With a small case study, we investigate on the feasibility of our approach and compare its performance against the barrier method. For this, we build a computationally-bound network service as a test bed for benchmarking the influence of live update on the end-to-end latency. In the server, a main thread listens for new connections and spawns a worker thread for each incoming connection. Upon a client request, the worker thread iteratively performs MD5 hash computations, starting from a client-provided initialization vector, until the first 20 bits of the hash digest become zero. The resulting hash, as well as its predecessor hash, are sent back to the client.

3.1 Evaluation Scenario

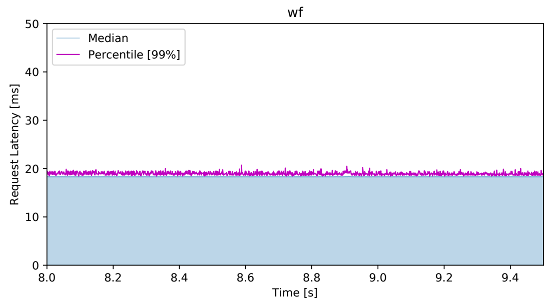
For our evaluation, we induced a bug into the server (see Figure 1), which we want to remove with a live update: The request handler (`handle_event()`), which is called by the worker threads, has an off-by-one error, due to a wrong comparison, in the hash-digest generation. Furthermore, we defined the quiescence point for the worker thread in their main loop; here, we either call `wf_migrate()` or synchronize all threads with the barrier method. We manually prepared a binary patch for the induced bug by extracting the correct function body for `handle_event()` from the fixed server binary. Both function bodies, the buggy and the correct one, are of equal length (1136 B), such that an in-place update is possible.

The measurements were performed on a setup with two recent desktop computers equipped with an Intel Core i5-7400 CPU (4 cores) and 32 GiB of memory. We connected both hosts by a switch via Gigabit Ethernet and used Ubuntu 18.04 LTS. One machine runs the server component, while the other is used as the client. On the server, we installed a modified Linux-5.1.0 kernel with our wait-free patching extensions.

To determine the server's response time, we measured the latency of each request on the client side: Four client connections are maintained during a benchmark run, in order to fully utilize all CPUs on the server. The clients repeatedly send requests with the same initial seed but with a random inter-request delay of 0 to 10 milliseconds. Each benchmark run has a total length of 16 seconds and starts with the buggy version of the server. One second after the server was started, we establish the client connections. Around 9 seconds after the run was started, we trigger the live patching of the server with a Unix signal.



(a) Barrier-Synchronized Patching



(b) Wait-Free Patching

Figure 4. Request Latencies of a Multi-Threaded Network Service during a Live Update. The benchmark was run 1000 times and we binned (bin size: 1 ms) the observed latencies on the time axis.

3.2 Request Latency

We conducted 1000 benchmark runs each for the barrier-synchronized method and the wait-free patching approach. Figure 4 shows the observed request latency values grouped into 1 ms bins along the time axis. The results for the barrier-synchronized patching (Figure 4a) and the wait-free patching (Figure 4b) benchmarks share a common baseline of a median request latency of about 18 ms.

At approximately 9 seconds[2] after the clients were started, we observe a latency spike in the barrier synchronization benchmark. This indicates blocked threads before the safe state is finally reached and the program is again able to progress. The median latency goes up to 30 ms and the 99%-percentil more than doubles its initial value. As expected, we do not see any latency change with the wait-free patching because the threads are able to seamlessly migrate, one by one, into the patched address space.

[2]The spike occurs shortly before the 9 seconds have elapsed. This is due to the non synchronized benchmark start times between the two computers and an initial client initialization delay.

3.3 Memory Overheads

Since our implementation doubles the memory requirement for the MMU configuration, we measured the memory overhead of our server application while running the benchmark. Immediately before the migration, the page-directory tree contains 27 internal nodes for the user space and 2161 for the kernel space. Because all kernel nodes are shared between all processes, they are only needed once for the entire system. This results in 108 KiB (27 pages · 4KiB) overhead for the server process. Since we applied the AS duplication technique for all processes in the system, the overall memory overhead for our prototypical implementation is the sum over all user spaces plus the kernel space. At the time of the benchmark, 50 processes required 1997 page-directory nodes, which results in 16.24 MiB of additional memory demand.[3] The total system memory consumption with buffer/caches included was 609.31 MiB.

4 Case Study: Memcached

Besides the latency comparison with the barrier method, we also validated the applicability of our wait-free approach with a real-world application. For this, we performed a live update of a running *memcached* [6] instance. We choose an actual bug-fix commit[4] that touched two functions: `drive_machine()` and `item_size_ok()`. We converted the commit into a suitable binary patch but eased the patch creation, which is not part of our contribution, with additional function padding in the unpatched version to account for grown function bodies.

The bug fix touches a central component of *memcached's* event-driven architecture: the state-transition function `drive_machine()`, which the worker threads repeatedly call for newly arrived requests or after an I/O operation has completed. With the bug fix, not only did the two function bodies change, but so did a jump table in the read-only segment, which the compiler generated for the central switch–case statement of `drive_machine()`. In total, the binary patch touched 11703 bytes in three places.

We are able to apply the live update with our wait-free method even under heavy request load (1 GBit/s) without observing any measurable difference in the request latency. However, we cannot report a quantitative comparison with the barrier method since we were unable to prepare a fair competitor that uses the barrier method so far. The reason is that, besides the worker threads, *memcached* also uses threads for log flushing and LRU-cache maintenance that perform sleep operations for up to a second.

[3]The address space is typically sparsely populated, hence the large overhead.
[4]Git commit hash: `32862944`.

5 Discussion

5.1 Costs and Benefits

A crucial aspect of the wait-free patching method is the lightweight address-space duplication (see Section 2). Most of the memory pages are shared. Only the changed regions and the MMU configuration has to be copied. This comes with benefits, like automatic zero-cost synchronization between the patch generations and the compatibility with atomic access. Concerning the MMU configuration, the current proof-of-concept chooses a rather simple approach. We only allow two active generations at a time and always allocate two adjacent sibling page directories which are synchronized at any time – even for kernel memory and for processes that do not use wait-free patching. Implementing on-demand address-space creation and synchronization could significantly reduce the memory footprint and would prevent unnecessary synchronization overhead. Furthermore, we could support more than two address-space generations. However, all this is associated with significant modifications in the Linux kernel, what we sought to avoid for the prototype.

A major benefit of the address-space duplication concept is that it is agnostic of the employed patching methodology. We can choose to use in-place patching (like in our case studies), but other strategies like function cloning or even replacing the whole text segment are also possible. In addition, the approach is flexible when it comes to data. Instead of sharing the whole memory, we can copy and transform parts of the program state – even in a lazy on-demand fashion.

5.2 Towards Wait-Free Dynamic Software Updates

We are aware that the presented method is only one component of a complete dynamic updating solution. In this paper, we intentionally focus on wait-free patching as an isolated approach without combining it with binary patch generation methods or state transfer techniques. Previous research has explored these topics in depth and created a variety of different approaches (see Section 6). We believe that wait-free patching based on address-space duplication can be combined with many of them, and we plan to investigate this possibility in the future.

5.3 Other Applications

Run-time binary modification is not limited to dynamic software updates. In a previous work, we presented Multiverse [22], a language-oriented approach for low-cost dynamic configurability. Based on annotated configuration variables, Multiverse generates specialized versions of functions and modifies the running system to use the version that matches the current configuration. From a technical perspective, Multiverse overwrites function call sites, which may be scattered throughout the whole program. In order to achieve a consistent update behaviour in programs with more than one thread, the patching procedure must be atomic with respect to all calls to the affected functions. Otherwise, two different *multiverse versions* could be executed at the same time while the patching procedure is in progress. Currently, Multiverse leaves this synchronization problem to the user. Our wait-free patching approach could be used to ensure atomicity by applying the modification in a new address-space generation, and then migrating the remaining threads. We have already carried out some initial tests, and we were able to confirm the feasibility of combining both approaches.

6 Related Work

The DAS [8] operating system incorporated an early run-time updating solution on module-level granularity. It requires absolute quiescence of a module to be patched. This is implemented by module-owned readers-writer locks and extended call & return operations. DYMOS [13] is a comprehensive language-oriented dynamic updating approach. It relies on the manual specification of conditions for safe code and data modification. The K42 [3] operating system exploits its strict object-oriented design to enable live kernel updates. The event-driven nature with short-lived and non-blocking threads makes it easy to define a safe state for concurrent patching, but it nevertheless relies on barrier synchronization. Proteos [7] is a research microkernel system with a process-level live update solution that focuses on automatic state transfer. Like our wait-free patching technique, it makes use of MMU-based address spaces, but unlike our approach the goal is not a seamless thread-by-thread migration. Instead, the whole process stops during the update procedure – the separate address space enables hot rollback.

Live-patching frameworks on function-level granularity [1, 2, 4, 5, 15, 19, 20] have become very popular because functions form a naturally bounded scope for changes, while still enabling relatively fine-grained updates. These approaches load a patched version of a function (function cloning) and install it via placing a trampoline jump at the beginning of the old function body (function indirection). Barrier blocking is the classical way to reach a global quiescence point where all threads are ready to be patched. But there are other possibilities: Ksplice [2] uses a polling technique where the whole kernel is repeatedly stopped and checked for a safe update state before the function indirection gets installed. This avoids long blocking delays but comes with the disadvantage that the patch may be applied late or never.

DynAMOS [15] and kGraft [19] extend the function indirection method: By placing additional redirection handlers between the trampoline and the jump target, they are able to decide on a per-call basis which version of a function (original/updated) should be used. The approach has some similarities to our address-space migration technique in that both methods are able to avoid blocking for a global safe state and instead perform a more sophisticated, context-dependent migration. However, the approach is limited to the function indirection method. Additionally, the trampoline may cause overhead and installing it may come with caching issues in SMP systems and poses additional difficulties on variable instruction-length architectures like x86.

LUCOS [4] 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. The virtualization layer is also used to enable state synchronization between the different versions of a function. POLUS [5] brought this idea to userspace and relies on the underlying operating system (ptrace & signals) instead of a hypervisor. Both approaches completely dismiss the idea of quiescence points and rely solely on state transfer which, to our knowledge, is not sufficient for all update scenarios. Furthermore, they are limited to patching on function-level granularity.

Ginseng [17] 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. It does not support multi-threaded programs.

Ekiden [11] and Kitsune [10] provide dynamic updates by replacing the whole executable code and transferring the program state. Updates are only possible at certain *update points* which also constitute synchronization barriers in the case of multi-threading. UpStare [14] 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 barriers. The authors suggest inserting barriers in long-lived loops and using non-blocking IO.

7 Conclusion

We present a wait-free approach for live patching that eliminates the operation disruption which is induced by the barrier-synchronized safe state requirements. We achieve this by duplicating the process's virtual address space and incremental thread migration. Thereby, threads explicitly switch at predefined quiescence points to the prepared address space, without transitioning through a waiting state.

With our prototypical implementation in the Linux kernel, we support live updating of code and read-only data. In two case studies with a simple network service and *memcached*, we show that our approach allows disruption-free operation during live patching, even under heavy load. Where barrier synchronized live patching provoked a request latency spike of +64 % (median), we could not detect a latency impact with the wait-free method in our experiments.

Acknowledgments

We thank the anonymous reviewers for their detailed and helpful feedback.

This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/3-1 and LO 1719/4-1.

References

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM '05)*. USENIX Association, Berkeley, CA, USA, 19–19.

[2] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: automatic rebootless kernel updates. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys '09)*, John Wilkes, Rebecca Isaacs, and Wolfgang Schröder-Preikschat (Eds.). ACM Press, New York, NY, USA, 187–198. https://doi.org/10.1145/1519065.1519085

[3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. [n. d.]. Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 USENIX Annual Technical Conference*. 279–291.

[4] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. 2006. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1134760.1134767

[5] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. 2007. POLUS: A POwerful Live Updating System. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 271–281. https://doi.org/10.1109/ICSE.2007.65

[6] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894

[7] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2013. 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)*. ACM Press, New York, NY, USA, 279–292. https://doi.org/10.1145/2451116.2451147

[8] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. 1978. Dynamic Restructuring in an Experimental Operating System. *IEEE Transactions on Software Engineering* SE-4, 4 (1978), 298–307. https://doi.org/10.1109/TSE.1978.231515

[9] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Proceedings*, Vol. 10379 LNCS. Springer-Verlag Italia, 161–176. https://doi.org/10.1007/978-3-319-62105-0_11

[10] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. 2014. Kitsune: Efficient, General-Purpose Dynamic Software Updating for C. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 13 (Oct. 2014), 38 pages. https://doi.org/10.1145/2629460

[11] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. 2011. State transfer for clear and efficient runtime updates. In *2011 IEEE 27th International Conference on Data Engineering Workshops*. 179–184. https://doi.org/10.1109/ICDEW.2011.5767632

[12] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. [n. d.]. NVthreads: Practical Persistence for Multi-Threaded Applications *(ECSC)*. ACM, 468–482. https://doi.org/10.1145/3064176.3064204

[13] Insup Lee. 1983. DYMOS: A Dynamic Modification System. www.cis.upenn.edu/~lee/mydissertation.doc

[14] Kristis Makris and Rida A. Bazzi. 2009. Immediate Multi-threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX '09)*. USENIX Association, Berkeley, CA, USA, 31–31.

[15] Kristis Makris and Kyung Dong Ryu. 2007. 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)*, Thomas Gross and Paulo Ferreira (Eds.). ACM Press, New York, NY, USA, 327–340. https://doi.org/10.1145/1272996.1273031

[16] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. 2014. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters* 9, 1 (2014), 526.

[17] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical Dynamic Software Updating for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 72–83. https://doi.org/10.1145/1133981.1133991

[18] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398.

[19] Vojtěch Pavlík. 2014. kGraft: Live patching of the Linux kernel. https://www.suse.com/media/presentation/kGraft.pdf, visited 2019-08-05.

[20] Josh Poimboeuf and Seth Jennings. 2014. Introducing kpatch: Dynamic Kernel Patching. https://rhelblog.redhat.com/2014/02/26/kpatch, visited 2019-08-05.

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

[22] Florian Rommel, Christian Dietrich, Michael Rodin, and Daniel Lohmann. 2019. Multiverse: Compiler-Assisted Management of Dynamic Variability in Low-Level System Software. In *Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM Press, New York, NY, USA. https://doi.org/10.1145/3302424.3303959

[23] M. Seltzer, V. Marathe, and S. Byan. [n. d.]. An NVM Carol: Visions of NVM Past, Present, and Future. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018-04). 15–23. https://doi.org/10.1109/ICDE.2018.00011