The Case for DBMS Live Patching

Michael Fruth
University of Passau
Passau, Bavaria, Germany
michael.fruth@uni-passau.de

Stefanie Scherzinger
University of Passau
Passau, Bavaria, Germany
stefanie.scherzinger@uni-passau.de

ABSTRACT

Traditionally, when the code of a database management system (DBMS) needs to be updated, the system is restarted, and database clients suffer downtime, or the provider instantiates hot-standby instances and rolls over the workload. We investigate a third option, live patching of the DBMS binary. For certain code changes, live patching allows to modify the application code in memory, without restart. The memory state and all client connections can be maintained. Although live patching has been explored in the operating systems research community, it remains a blind spot in DBMS research. In this *Experiment, Analysis & Benchmark* article, we systematically explore this field from the DBMS perspective. We discuss what distinguishes database management systems from generic multi-threaded applications when it comes to live patching. We then propose domain-specific strategies for injecting quiescence points into the DBMS source code so that threads can safely migrate to the patched process version. We experimentally investigate the interplay between the query workload and different quiescence methods, monitoring both transaction throughput and tail latencies. We show that live patching can be a viable option for updating database management systems, since database providers can make informed decisions w.r.t. the latency overhead on the client side.

PVLDB Reference Format:
Michael Fruth and Stefanie Scherzinger. The Case for DBMS Live Patching. PVLDB, 17(13): 4557 - 4570, 2024.
doi:10.14778/3704965.3704966

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sdbs-uni-p/vldb25-dbms-live-patching.

1 INTRODUCTION

Database management systems (DBMS) are part of the critical IT infrastructure and must be maintained with care. When it comes to security patches, database clients may experience database restarts as highly disruptive, especially when long-standing connections are severed. At the very least, restarts can be untimely and force database clients to work around downtimes.

Considerable effort has been made to accelerate the restart of DBMS servers [6, 22, 57]. Facebook, for instance, relies on *shared memory* to accelerate the restart of certain distributed systems [1, 5, 39]: Among these systems is Scuba [1], a main memory database backing about 120 GB, for which the restart time was reduced from 2–3 hours to 2–3 minutes [22]. While this may be acceptable for a full upgrade, it may not be justifiable for a small (security) fix. Alternatively, additional DBMS instances are run in parallel (e.g. as hot-standby or multi-master) [9, 17, 32, 33, 36, 37, 43], and updates are realized by *rolling over* [3, 8, 12, 17, 31, 35, 44] on these instances.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704966

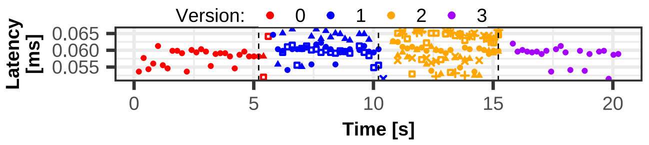


Figure 1: Seamlessly live patching MariaDB through four code versions. Colors distinguish the code versions.[1]

Motivation. We investigate a *third way*, made possible by recent advances in live patching user-space applications for Linux. Live patching performs a code change (i.e., a *patch*) directly in memory, while the software is running. Instead of a restart, the threads are gracefully patched so that they read from an updated code segment.

We illustrate the potential of this approach for DBMSs with a micro-experiment using the WfPatch framework [48]. We run a version of MariaDB with *quiescence points* injected into the application source code. When the control flow of a thread reaches a quiescence point and a patch exists, the thread migrates to the patched version. Figure 1 shows the query latencies, while the number of connections is scaled up and down. Each data point is one measurement. Every five seconds, we initiate live patching to migrate the DBMS binary to a new version. The colors indicate the individual versions, and the shapes distinguish the database connections (allowing to discern new connections). As can be seen by the change in color, MariaDB gracefully migrates through four code versions while maintaining all existing connections, evaluating queries, and even accepting new connections.

State of the Art. In the maintenance of operating systems, live patching is established practice: IBM AIX [28], Windows virtual machines for Azure [34], or different live patching tools for Linux (e.g. Kpatch [42] from RedHat, kGraft [53] from SUSE or Ksplice [4] from Oracle) are routinely used in production environments, showing that live patching is feasible, even for highly critical infrastructure.

While live patching in the kernel space is state-of-the-art, live patching applications in user-space is still underdeveloped. Any tools publicly available are limited to research prototypes. Proof-of-concept evaluations often include database management systems [24, 30, 48, 56]. Yet generally, the DBMS software is merely

[1]Throughput per worker fixed at 5 queries per second. Extreme latencies below 10th / above 90th percentile filtered out to improve the readability of the chart. Patching using local quiescence for the one-thread-per-connection policy, concepts to be introduced. Patching with a synthetic patch, details provided with our artifacts.

treated as yet another generic multi-threaded application, the same as caches or web servers. Thus, the special challenges of database management systems are not adequately taken into account.

Notably, several commercial database providers advertise live patching capabilities. For example, Azure allows live patching of the SQL Server Engine. A blog claims that more than 80% of typical SQL bug fixes can be applied by live patching [40]. This indicates the vast potential for practical impact of database live patching. However, since these tools are exclusively operated in-house, they are not available for experimental analyses.

So far, there has been no systematic exploration of the potential of live patching specifically for multi-threaded DBMS software from the unique perspective of database systems research.

Contributions. In this paper, we systematically explore the feasibility of live patching for database management systems.

- DBMSs have unique characteristics that set them apart from other multi-threaded applications. We show that there are specific desiderata for live patching to be practical. In particular, we focus on how to statically prepare the code of database connection management; we study two common policies, one-thread-per-connection and thread pools.
- We propose a novel approach for achieving safe thread quiescence (the prerequisite for live patching) in database thread pools. Our solution defines the order in which threads enter into quiescence based on the role of the thread (e.g., listener or worker thread). We empirically show that our approach operates without deadlocks.
- For our extensive experiments, we prepared two open source DBMSs for live patching (MariaDB and Redis). We successfully apply real-world patches from GitHub and explore two alternative quiescence methods. We experimentally evaluate live patching from different stakeholder perspectives and identify the key factors determining the performance. In particular, we study the impact on extreme latencies under different query workloads. Our insights enable database providers to make an informed decision w.r.t. live patching.

2 DBMS-SPECIFICS

Desiderata. In patching a multi-threaded application, we prefer a short *synchronization time*, i.e. the time it takes from patch triggering until *all* threads run in the patched version. Ultimately, this determines how quickly a security vulnerability can be closed.

However, database management systems constitute a family of applications with highly specific requirements that set them apart from other multi-threaded applications. We formulate these additional desiderata for patching a DBMS software binary:

(1) The DBMS server maintains the existing client connections, and even allows for new connections to be made.
(2) Patching does not cause database deadlocks.
(3) The database state (which can be large) remains available.
(4) Code patches can be applied for different query workloads.

Not all of these desiderata can be met with conventional update methods: In a classic system restart, connections must be severed, transactions aborted, and the database state must be restored upon restart (costing minutes or even hours [22, 57]). When running instances in parallel, existing client connections must be carefully handled, and a short failover time may still be noticeable. Throughout, the hardware requirements multiply (temporarily).

In the following, we discuss the above desiderata in light of live patching and point out technical challenges.

Technical Challenges. (1) Implementations for database connection handling range from simplistic to complex: The key-value store Redis uses a single-threaded event loop, where all commands are executed in sequential order (since version 6, Redis supports multiple threads for I/O). In contrast, PostgreSQL maps each connection to its own process and forgoes multi-threading. MariaDB is multi-threaded and supports different connection management policies. Changing the DBMS source code is system-specific and requires extensive domain knowledge.

(2) Implementing a transaction system is a delicate task. Any manipulations of the source code that cause threads to block (as quiescence points ultimately will), amplify the risk of deadlocks. Again, developers must be highly prudent.

(3) Compared to other families of applications, database management systems can hold very large states in memory. These states are expensive to recover at system restart and may also be expensive to copy during live patching, where we need to prepare the new address space containing the code changes.

(4) Different query workloads bring about different challenges. In particular, workloads containing long-running queries are likely to be unsuitable for live patching methods in which all threads must block until they reach a global barrier. Such a “stop the world” event may even noticeably increase extreme latencies.

Stakeholders. In exploring live patching for DBMS, we assume different stakeholder perspectives.

From the *perspective of the developer* of the DBMS, quiescence points must be injected into the source code in a safe manner. Specifically, changes in the system must not introduce new deadlocks. Moreover, changes to connection management must not noticeably deteriorate query throughput. Note that these extensions to the DBMS source code are a one-time effort (although, of course, these code changes must be maintained over time).

From the *perspective of the database clients*, performance should not degrade. Ideally, database clients remain unaware that patches are being applied. Especially in distributed settings, (tail) latencies are a particular concern, since latencies exceeding the 99th latency percentile can degrade the client experience (and when they build up, even the entire system performance [13]).

We also assume the *perspective of the database provider*, who has to choose between performing a restart of the application, rolling over on standby instances, or live patching. This requires that decision makers be able to predict the latency overhead. Consequently, we explore the key factors that determine the patch application time, such as the size of the database state and the size of the patch.

Scope of this work. We focus on the technical aspects of live patching multi-threaded DBMS, but not the question of whether the behavioral changes of a code change make it suitable for live patching. As described in Section 3.3, there are certain technical constraints to consider. Yet ultimately, deciding whether the behavioral changes of the code allow for live patching a database running in production warrants separate research and is beyond the scope of this article.

3 BACKGROUND – LIVE PATCHING

There are several (prototypical) tools for live patching user-space applications and we refer to our discussion of related work for an overview (Section 4). In our introduction to the core concepts, we focus on the framework WfPatch [48] by Rommel et al.

3.1 Quiescence Points

Live patching changes the memory state of a process. For this to be safe, a thread must be in a well-defined state. WfPatch relies on software developers to identify such safe states and impose barriers, the *quiescence points*, in the application source code. Once the control flow of a thread reaches a quiescence point, the thread is either blocked, patched, or continued. The action depends on the chosen quiescence method, and we discuss two methods below.[2]

3.2 Quiescence Methods

Global. A thread blocks when it reaches a quiescence point. Once all threads have reached their quiescence point, the patch is applied, and all threads collectively migrate to the patched version.

The upper half of Figure 2 visualizes a scenario with global quiescence. We assume a DBMS with one background thread, two threads each serving a connection, and one patcher thread. The patcher thread is spawned by WfPatch and performs all heavy-weight tasks for patch application (see Section 3.4.2). At time t_{G1}, a patch request is made. Upon completing task $T2$, the background thread blocks. Then the thread serving connection 1 blocks. At time t_{G4}, the thread serving connection 2 reaches its barrier, achieving global quiescence. The patcher thread applies the patch. By time t_{G5}, all threads resume work in the patched version.

Drawbacks. In global quiescence, blocking threads can cause various problems, as exemplified next: (1) Long wait times: All threads reach their quiescence point timely except the thread serving connection 2. If it executes OLAP-style queries, long wait times occur. (2) Unbound wait times: Similar to problem 1, but now connection 2 is idle. It waits for user input to reach its barrier, causing unbounded wait times. (3) Deadlock: Connection 1 blocks at its barrier while holding a lock. Connection 2 is waiting for the release of this lock, resulting in a cyclic dependency and, therefore, a deadlock.

Local. In local quiescence, each thread can individually migrate to the patched process version, upon reaching its quiescence point. No blocking or waiting for other threads is needed.

Local quiescence is visualized in the lower half of Figure 2: At time t_{L1}, a patch request is made, and the patcher thread prepares the patched process version. At time t_{L2}, the patched process version is ready and threads can migrate to this version. The thread serving connection 2 and the background thread both reach a quiescence point at time t_{L3} and migrate to the new version. Finally, the thread serving connection 1 reaches its barrier at time t_{L4}. By time t_{L5}, all threads run in the patched process version.

3.3 Categorizing Patches

From a technical point of view, each change in source code affects a different region of the memory layout of a program. Not every patch can be applied with every live patching framework, as these are often restricted to patches that affect certain memory regions (more details in Section 3.4.3).

[2]Rommel et al. further propose group quiescence, which we do not consider here.

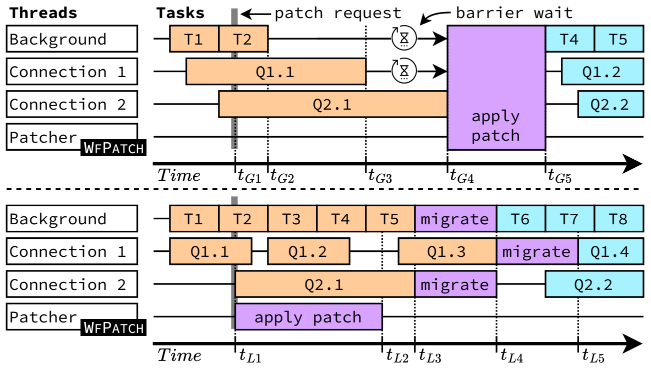


Figure 2: Live patching a multi-threaded DBMS: global (top) vs. local quiescence (bottom). Based on Figure 2 of [48].

Furthermore, the semantics of a patch, i.e. the effect or changed behavior of the application after patching, must be well understood. While fewer problems arise due to the non-blocking property of local quiescence, this method is not universally applicable and restricted to a certain category of semantic changes. In discussing this next, we categorize patches according to their effect.

A *thread-local patch* is a code change that affects only the given thread itself and no other threads. For example, consider a security fix that adds a boundary or a NULL pointer check [27]. Thread-local patches can be applied with any quiescence method, including local quiescence. In the case of a *thread-group patch*, the changed or patched behavior affects some threads, but not all. For example, consider that the data to be processed changes in a producer-consumer scenario [27], so a joint migration is essential. This demands global quiescence. Finally, a *process patch* enforces that all threads of the process are patched at the same time. This is also only feasible under global quiescence.

Statically determining the correctness of dynamic updates is an undecidable problem [23]. Only a skilled developer can judge the effects of a given patch and categorize it accordingly.

3.4 WfPatch Framework

The WfPatch framework consists of a modified Linux kernel, a user-space library, and a customized version of Kpatch [42] for patch generation. We explain the concepts necessary to understand this article and refer to the original article for further details [48].

3.4.1 *Kpatch.*

Kpatch [42] is a suite of tools for live patching the Linux kernel. Rommel et al. [48] customized the Kpatch tool to also support patch generation for user-space applications. The modified version of Kpatch is used for patch generation, while loading and applying the patch is handled by the WfPatch user-space library.

3.4.2 *Address Space Generation.*

With global quiescence, a patch is applied *directly* to the address space of the process. However, with local quiescence, multiple address space generations are managed. We outline these concepts in the following.

Linux is divided into user-space and kernel-space: All applications of the Linux kernel run in kernel-space and application software etc. run in user-space (e.g. a DBMS). The memory, also called *address space (AS)*, of a user-space application is shared between its threads. For example, the stack, heap, or .text segment (executable instructions) reside in the address space. On a low-level basis, the address space consists of a number of regions or *virtual memory areas (VMAs)*. A VMA forms a contiguous memory area. Each VMA is further divided into several pages, representing the smallest unit (the typical size is 4096 B). All access to memory is performed on virtual addresses which are translated based on page tables to physical addresses. Each process contains information about its address space, i.e. a list of VMAs, the page table, etc. This information is organized within the *memory map (MM)*. Thus, an address space is the abstract concept represented by the MM structure in Linux.

Linux has a strict one-to-one relationship between the memory map and the process. WfPatch relaxes this so that threads of the same process can have different memory maps, i.e. threads can operate in different address spaces, yet the individual memory maps remain siblings. When creating a new memory map, the memory map data structure with all its attributes of the calling thread is copied and kept in sync with its siblings by sharing pages. Logically, they are two separate address spaces, but all entries refer to the same pages. All memory maps are kept in sync using the first memory map as a synchronization point. Each distinct memory map forms an AS generation. Synchronization between AS generations can be stopped on the level of individual VMAs. The *copy-on-write (COW)* mechanism is used on VMAs that are no longer shared. Using COW, changes are no longer reflected in the other AS generations, as the page is copied when modified.

The way WfPatch clones a memory map is similar to the fork() system call: A new process is created by duplicating the AS of the calling process. All pages of both processes are shared as read-only and marked as COW. However, there is a difference from AS cloning: All pages are shared by default (shared mapping), and changes are synchronized with all other AS generations. Only certain VMAs of the AS are marked as COW.

Once an AS has been created and the corresponding regions have been marked as read-only, a patch can be applied to create a patched AS generation: WfPatch loads the patch binary file generated by Kpatch, extracts all sections, and applies them to the current AS.

The patch binary in Executable and Linking Format (ELF) lists changed sections between the unpatched and patched object files of the application to patch (for details about the patch-structure, we refer to the "create-diff-object" utility in [15]).

WfPatch Operations. Figure 3 illustrates a process having two AS generations (initial AS on the left; cloned AS on the right) and five threads. Each box in memory represents a page, whereas an arrow pointing to it represents the translation process of the page table (gray bar next to the AS). The fill color of a box for the physical memory defines its content, while virtual memory boxes are colored for easier navigation. The steps and the required WfPatch operations to achieve the illustrated state are labeled with circled numbers and the respective operation.

Before the AS in Figure 3 is cloned, the pages in the region of addresses 4–11 are defined as read-only shared mapping (❶ `wf_pin()`) because, for example, we assume that the .text and .rodata segments

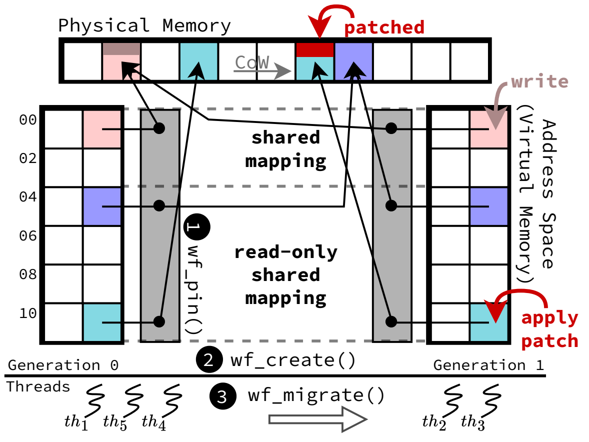


Figure 3: Process memory layout after WfPatch operations.

are located there. All other pages remain as shared mappings. Subsequently, the AS is cloned (❷ `wf_create()`). The patch is applied to the new AS and affects the page at address 11, which is in the area of the read-only shared mapping. This results in the physical page to be copied. The change is ultimately applied to the copied page. AS generation 0 still points to the old unpatched physical page. Next, two threads migrate to the new AS generation (❸ `wf_migrate()`). To highlight the shared mapping, a write to the page at address 1 is reflected by both AS generations, as both still point to the same physical page. The workflow of WfPatch is to (1) clone an AS, (2) apply a patch, and (3) individually migrate threads to the patched process version.

3.4.3 *Technical Limitations.*

Kpatch can only generate patches for applications written in C. Furthermore, the granularity of patches is based on functions, i.e. old functions are replaced with new ones. Additionally, a patch can only be applied to inactive functions, i.e. functions that are not currently active on the stack frame.

WfPatch can only patch the read-only regions .text (executable code) and .rodata (initialized static constants). Thus, global variables or the layout of data structures cannot be patched.

Despite these limitations, a large share of patches can be applied in practice. Rommel et al. [48] show in an analysis of more than 100 software fixes for six applications that about 87% of the patches affect only the .text segment.

4 RELATED WORK

We first discuss live patching in the operating systems community. These contributions, to the best of our knowledge, have not yet been systematically explored in database systems research.

Live Patching Research in the OS Community. Various (prototypical) user-space live patching frameworks exist [7, 24, 25, 30, 38, 48, 52, 56]. Several enforce some form of global quiescence [24, 25, 52], while others allow patch application at arbitrary points in time, but with the overhead of halting all threads [7, 30].

To our knowledge, WfPatch [48] is the only framework for live patching multi-threaded user-space applications with wait-free patch application via local quiescence. The idea of multiple AS generations of WfPatch has found further applications, including thread-specific security [50] and execution variations [54].

Under the hood, WfPatch and Kpatch [42] employ trampolines, a common technique used in live patching [2, 4, 7, 48, 53] to redirect function calls to the patched code once the control flow reaches it.

Live Patching DBMS in the OS Community. Database systems are part of experimental evaluations of several live patching approaches [24, 30, 48, 56], but commonly treated as generic user-space applications, the same as caches or web servers. The unique challenges of DBMSs are not considered, and the database client experience is largely ignored. This can also be observed in the experiments of Rommel et al. [48]: The authors evaluated the WfPatch framework based on six different multi-threaded user-space applications, including MariaDB (the only relational DBMS in their work). In the following, we discuss their experiments with MariaDB from the perspective of database systems, which emphasizes aspects distinct from those prioritized in operating systems research.

Request Latencies. Rommel et al. executed a customized benchmark against MariaDB with the one-thread-per-connection policy (thread connection policies are explained in Section 5). Quiescence was triggered every 1.5 seconds, but without actually applying a patch. This experiment was performed for global and local quiescence, and the results for both quiescence methods were compared based on a histogram of measured client request latencies.

Runtime Penalty. Rommel et al. reported the runtime penalties of WfPatch operations. They measured the overhead of AS cloning and AS switching for a single, fixed MariaDB configuration.

Discussion. Both experiments focus on WfPatch and the concept of local quiescence in comparison to global quiescence. From the perspective of database systems research, additional aspects should be included: (1) In addition to the one-thread-per-connection policy, MariaDB also supports a thread pool policy. (2) Database systems are subject to different types of workload, such as OLTP or OLAP. (3) The steps of loading and applying a real-world patch are not captured by the experiments. (4) Different kinds of database systems (e.g. main memory vs. disk-based) have memory states of different size, and also differ in how they store data internally.

Assuming the domain-specific perspective of database systems research, different experiments and analyses are required to assess live patching DBMSs along the desiderata outlined in Section 2. Given that their quiescence points for the thread pool connection policy are susceptible to deadlocks (see Section 6.1.1), we propose a novel approach known as priority-based quiescence (detailed in Section 5.2), which aims to ensure a safe and deadlock-free migration of threads within a thread pool.

DBMS Upgrade Strategies. A common approach to apply a patch to a DBMS without database clients noticing downtime is to perform a rolling upgrade, based on running additional instances [3, 8, 12, 17, 31, 35, 44]. In this setup, the hardware costs multiply due to redundant provisioning of hardware and database instances. Furthermore, a rolling upgrade may, nevertheless, take time for a cluster to patch instance-by-instance. Furthermore, cluster performance is reduced during downtime, and the instance needs to recover its full memory state on startup.

Live Patching DBMS in the Database Systems Community. Research on live patching database systems is still in an exploratory stage. In a very early-work abstract, we conducted a first experiment on live patching a multi-threaded DBMS [18]. Since then, we have systematically extended our work of specifically addressing challenges inherent to multi-threaded, single-instance databases, to the point where we propose a novel solution for live patching with database thread pools. Moreover, in another research we expanded our focus to *distributed* database systems, exploring live patching for distributed in-memory key-value stores [19] using Redis Cluster as a reference system. We developed and evaluated patch distribution strategies and proposed guidelines for enabling live patching in distributed databases. To validate, we implemented these guidelines for a primary-replica PostgreSQL setup. This research specifically focuses on the distributed aspects while not addressing multi-threaded concerns, as Redis Cluster nodes are primarily single-threaded and PostgreSQL is multi-processed.

According to a blog entry [40], Azure SQL Database supports live patching since 2018. It employs an optimized C++ compiler for patch generation, and uses trampolines to redirect function calls to patched code. In particular, this approach is designed for Windows systems (whereas WfPatch works for Linux). Furthermore, only few details about the solution for Azure SQL Database are known, and we found no published systematic experiments.

DBMS Address Space Optimization. The novelty of WfPatch comes from duplicating an address space. Its functionality is similar to Linux fork(), which is a common operator in today's database landscape to perform a snapshot of the memory state. Different database systems try to work around the overhead of fork() by, for example, reducing the number of page table entries by using larger page sizes [26]. A recent proposal of asynchronous fork [41] has been made to also reduce the fork() overhead of Redis [47].

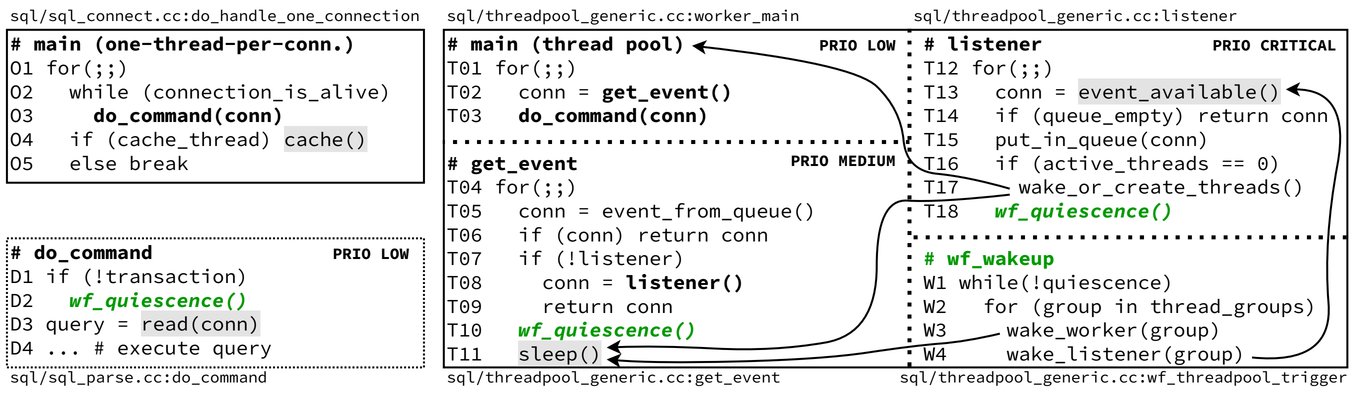
The exploitation of the address space or virtual memory has also found application in other areas, such as caching (e.g. DBMS buffer pool) [29] or query processing [51].

Priority Scheduling in DBMSs. Our novel contribution of priority-based quiescence designed for database thread pools, to be introduced in Section 5.2, is independent of other priority mechanisms used within a database system, such as task scheduling [45, 46]. In fact, our priority-based quiescence concept can seamlessly integrate with other mechanisms. For example, by aligning the priorities of quiescence with task priorities, it could be ensured that threads engaged in high-priority tasks are allowed to execute for at least as long as there are threads performing lower-priority tasks.

5 SAFE QUIESCENCE POINTS IN DATABASE CONNECTION MANAGEMENT

Database connection management is a carefully tuned component, and developers need to take great care when injecting quiescence points. They must ensure that quiescence points are indeed safe states for threads to migrate. Ideally, the quiescence points do not noticeably alter the connection management policy, and the same quiescence points can be used with both global and local quiescence.

Next, we discuss the established connection management policies one-thread-per-connection and thread pool. We first consider quiescence points w.r.t. *global quiescence*, which has the strictest requirements, and then discuss local quiescence.



```
sql/sql_connect.cc:do_handle_one_connection
# main (one-thread-per-conn.)
01 for(;;)
02   while (connection_is_alive)
03     do_command(conn)
04   if (cache_thread) cache()
05   else break
```

```
# do_command                        PRIO LOW
D1 if (!transaction)
D2   wf_quiescence()
D3 query = read(conn)
D4 ... # execute query
sql/sql_parse.cc:do_command
```

```
sql/threadpool_generic.cc:worker_main
# main (thread pool)                PRIO LOW
T01 for(;;)
T02   conn = get_event()
T03   do_command(conn)
# get_event                      PRIO MEDIUM
T04 for(;;)
T05   conn = event_from_queue()
T06   if (conn) return conn
T07   if (!listener)
T08     conn = listener()
T09     return conn
T10   wf_quiescence()
T11   sleep()
sql/threadpool_generic.cc:get_event
```

```
sql/threadpool_generic.cc:listener
# listener                     PRIO CRITICAL
T12 for(;;)
T13   conn = event_available()
T14   if (queue_empty) return conn
T15   put_in_queue(conn)
T16   if (active_threads == 0)
T17     wake_or_create_threads()
T18   wf_quiescence()
# wf_wakeup
W1 while(!quiescence)
W2   for (group in thread_groups)
W3     wake_worker(group)
W4     wake_listener(group)
sql/threadpool_generic.cc:wf_threadpool_trigger
```

Figure 4: Implementing one-thread-per-connection (top left) and thread pool policy (right block), inspired by MariaDB.

5.1 One-Thread-per-Connection with Global Q.

Preliminaries. To introduce the one-thread-per-connection policy, we walk through a pseudocode implementation. While our pseudocode is inspired by MariaDB source code[3], the considerations in setting quiescence points are applicable in a more general context.

To the top left, Figure 4 shows the main function of a worker thread. For now, we ignore the commands and functions set in italic green, which marks the injected code. Each connection is assigned a dedicated thread, its *worker*. The worker executes commands, such as queries (line 03), as long as the connection to the client is *alive* (line 02). Once the client connection is closed, the worker is either *cached* (line 04) or terminated (line 05). The do_command function is the starting point for query processing: First, it performs a blocking read on the client connection (line D3) and waits for input from the client. Once input is available, the query can be executed (line D4).
Challenges. When injecting quiescence points, it is essential to avoid deadlock and starvation. (1) *Deadlocks:* A thread that encounters a quiescence point will block and wait for the other threads to reach the barrier. If this thread already holds a lock to a data object, this can cause a deadlock when another thread requires this lock before it is able to reach its own quiescence point. (2) *Starvation:* A thread that remains cached will not reach its quiescence point and therefore blocks all other threads that already wait at their barrier.
Solution. The solution presented here is based on the approach proposed by Rommel et al. [48] (with minor refactoring) and addresses the following challenges: (1) To not increase the risk of deadlocks, a worker thread must be outside of a transaction when it encounters a quiescence point. Therefore, we check the transaction status (line D1 in Figure 4) before a quiescence point is reached (line D2). It is generally best practice to inject quiescence points high up in the call hierarchy, as only functions that are not currently active on the call stack can be patched at runtime. (2) To prevent starvation, a patch request must cause all cached threads to wake up so that they may then reach their quiescence point. This wake-up call is triggered by the patcher thread (code not shown).
Discussion. An inherent problem concerns blocking reads, where a thread waits for user input. In global quiescence, this will cause unbound wait times (line 03), for example, with a client holding an idle connection. This problem can be addressed by intentionally interrupting the thread, as we also do with the listener in the thread-pool policy (discussed next).
Implementation. We adopted the solution based on Rommel et al. [48] and implemented it with minor changes in MariaDB.

5.2 Thread Pool with Global Quiescence

Preliminaries. In the thread pool as implemented in MariaDB, thread groups are used to partition client connections. The size of the thread pool corresponds to the number of thread groups, whereby each thread group can consist of several threads with different roles (this is explained in more detail below). In the following discussion, we may safely assume that there is only one thread group, as the thread groups operate independently of each other.

We first assume that the database workload is high. Then, a dedicated listener thread manages a queue to distribute the work among the worker threads. Consequently, the listener adds connections to the queue when they have work available (lines T13 and T15 in Figure 4). Based on the producer-consumer model, worker threads dequeue connections from the queue (line T05) and then process the query (lines T06 and T03). The dedicated listener thread is only active in the `listener` function. With each iteration, it is checked whether an active thread can process the previously added event (line T16). Otherwise, a sleeping thread is awakened or a new one is created (line T17; see arrows in Figure 4). If the work queue is empty, the consuming worker thread goes to sleep (line T11).

For a medium-to-low workload, there is no dedicated listener thread, but the worker threads temporarily assume this role: A worker transitions to listener (line T08) and waits for a connection to become available to process data. Once a connection has input, the input is fetched (lines T14 and T09) and processed (line T03).
Challenges. Since this policy is more complex, the risk of accidentally introducing deadlocks or allowing starvation is amplified.[4] The quiescence points in which a thread pool has no dedicated listener are similar to the one-thread-per-connection policy. Specifically when MariaDB faces high loads and the thread pool has a dedicated listener, we must carefully control the order in which threads may block upon reaching their quiescence points.

[3]Based on git hash: 06fae75859. The names and functions shown as pseudocode differ slightly from the original implementation, they were edited for easier readability.

[4]Rommel et al. provide an implementation for the thread pool that runs into deadlocks in our experimental setup. This highlights the challenge in finding a functional solution.

Let us illustrate these risks and consider the scenario of a worker thread inside an active transaction waiting for the release of a lock.

Active Worker vs. Listener. A listener thread reaches its quiescence point and blocks. It no longer manages the queue. As long as the event which could release the desired lock is not added to the queue, the worker thread cannot complete its transaction. As quiescence points are purposefully positioned outside of transactions, the worker thread is indefinitely blocked.

Active Worker vs. Sleeping. A worker that awakes from sleep may have to be prevented from blocking when it encounters a quiescence point (line `T10`). Otherwise, there may not be a worker left that handles events entering the queue, which could release the desired lock of the active worker. This constitutes a deadlock.

Novel Solution – Priority-Based Quiescence. These scenarios motivate us to propose *priority-based quiescence*, a priority-based approach to orchestrate blocking of threads. We assign priorities to threads depending on their current role, where threads with higher priority will only block at their quiescence point *after* all lower-priority threads block. Put differently, a thread encountering a quiescence point skips it if there is still a lower-priority thread that has not yet reached its quiescence point. Intuitively, the higher-priority thread has not yet reached a state where blocking is *safe*.

General Applicability. To adopt this approach, it is essential to identify the specific roles or tasks that a thread can perform. Prioritization is established on the basis of the following hierarchy: Tasks relying on others are assigned lower priorities, whereas tasks that are prerequisites for other tasks to proceed are given higher priorities (the hierarchy is reflected in the priorities). As a result, threads responsible for tasks on which other threads depend remain active until all these dependent threads are blocked.

Following the assignment of priorities, careful consideration is required when placing quiescence points. Integration should guarantee that each thread consistently encounters quiescence points. Threads should only pass a quiescence point when they do not hold locks on shared resources. In scenarios with blocked or sleeping threads, common in a thread pool setup, an external mechanism (triggered by the WfPatch thread) can awaken them, ensuring a reliable progression toward the quiescence point.

Adoption to MariaDB. In Figure 4, the priority of each role is annotated to the top right of each code block. For MariaDB, three roles are identified with their respective priority: active worker (LOW), sleeping worker (MEDIUM) and listener (CRITICAL). The listener blocks last since it accepts incoming data upon which both workers depend. A sleeping worker is awakened to handle queries, supporting active workers in completing their tasks. Consequently, a sleeping worker should only block after active workers.

We have also injected a dedicated quiescence point for each role of a thread (line `T10` for a (sleeping) worker, line `T18` for a listener). To avoid problematic scenarios between blocked workers and sleeping workers, we trigger the `wf_wakeup` method from the outside. It wakes all sleeping worker threads (line `W3`) and blocking listener (line `W4`) until global quiescence is reached (line `W1`).

Discussion. The priority-based scheme is designed to prevent the deadlock scenarios outlined above.

Implementation. We implemented our novel concept of priority-based quiescence for thread pools in MariaDB and extended the WfPatch user-space library to support priorities.

5.3 Adaption to Local Quiescence

For both connection policies, the same quiescence points described above can also be utilized for local quiescence. We do not require adaptation, since local quiescence is not plagued by the problems of global quiescence (bound/unbound wait times, deadlocks). In fact, we could inject additional and "local quiescence specific" quiescence points in the source code. However, these benefits come with the limitation of reduced patchability, since local quiescence is limited to thread-local patches. Since global quiescence has the stricter requirements and local quiescence is compatible, we settle on the shared set of quiescence points in favor of less code complexity.

6 EXPERIMENTS

We evaluate live patching for database systems from the stakeholder perspectives discussed in Section 2. For enlarged and additional experiment plots, we refer to the extended version of this article [20].

Hardware. Our server has two Intel Xeon Gold 6248R CPUs (24 cores per CPU; 3.0 GHz) and 384 GB of main memory. To reduce system noise, Intel Turbo-Boost is disabled. All CPU cores run at a fixed core frequency of 3.0 GHz. Since we assign more than twice the number of cores to each application (DBMS / benchmark framework) as there are concurrently running queries (detailed configuration given below), we have disabled Intel Hyper-Threading to utilize all 24 physical cores per CPU and to avoid competition for shared core cache.

Live Patching Infrastructure. The system runs Debian 11 with the latest WfPatch[5] Linux kernel (version 5.15) at the time of writing. Live patching also requires the WfPatch user-space library.

Applications to be patched. We extended the source code for the RDBMS MariaDB and the key-value store Redis. MariaDB implements the connection management policies of interest, and Redis allows us to easily control the size of the memory state.

***MariaDB.* Code Extensions.** We extended the MariaDB source code as discussed in Section 5. We injected quiescence points for the main thread, all worker threads, and dedicated listener threads. Therefore, we cover all threads relevant to transaction processing.[6]

Patches. We developed a fully automated pipeline to analyze the development history of an application for live-patchable code changes. For every commit, we check whether Kpatch can generate a patch and, upon success, automatically apply our source code changes, specifically injecting the quiescence points. In this way, we obtain a wide range of patches and their various characteristics.

The Kpatch tool faces limitations in generating patches for MariaDB, as MariaDB is written in C/C++ and Kpatch targets C. Despite this, our automated pipeline, scanning versions 10.5.0–10.5.13 of MariaDB on GitHub, identified 117 live-patchable code changes. While all 117 patches contribute to our broader analysis (see Section 6.3), we imposed two strict filters for our in-depth evaluation: (1) the patch must be officially labeled a "bug" in the MariaDB bug tracker, and (2) it should modify a function in the stack trace below the `do_command` function. In consequence, the patch actually affects a function that is executed during a benchmark run (and not some dormant code, which is low risk to patch). The five selected patches

[5]https://github.com/luhsra/linux-mmview; git tag: `mmview-v5.15`

[6]There are further threads in MariaDB which we do not live patch in our experiments. This is merely a technical limitation that can be easily resolved in productization.

Table 1: MariaDB patches fixing official bugs.

ID	git hash	MariaDB Jira	#LoC added	#LoC deleted
#1	18502f99eb	MDEV-22185	+1	-1
#2	30d41c8102	MDEV-22881	+2	-1
#3	3bb5c6b0c2	MDEV-22113	+7	-8
#4	56402e84b5	MDEV-21824	+1	-1
#5	5b678d9ea4	MDEV-25251	+1	-1

are presented in Table 1. The first column provides a unique identifier that is used throughout this paper. In the PDF, the git hash and the corresponding Jira ticket of the code change are clickable links, pointing to the respective entries on GitHub and Jira, so that readers may inspect them in detail. The last two columns show the number of lines changed, excluding tests. Note that these patches resolve real-world bugs by changing just a few lines of code.

Benchmarks. We adapted the benchmark harness BenchBase[7] (formerly OLTP-Bench [14]) to trigger patch application. For OLTP workloads, we use the benchmarks NoOp, YCSB [11] (scale factor 1,200) and TPC-C [55] (scale factor ten). NoOp (No Operation) is extremely lightweight and just sends a single semicolon to the database. For NoOp, we run BenchBase with the Epsilon Garbage Collector, as Java garbage collection can interfere with latency measurements [21]. For all OLTP benchmarks, a ten-second warm-up phase is followed by a 30-second benchmark phase.

For OLAP-style queries, we removed all OLTP queries from the benchmark CH-benCHmark [10]. BenchBase is configured to execute a 30 minute measurement phase without warm-up phase, and to trigger the patching process after five minutes.

Configurations. For all benchmarks against MariaDB, we use ten terminals, i.e. ten parallel connections. MariaDB is executed with default settings, except for the thread pool which is limited to three thread groups. Connections are assigned round-robin to the three groups, each consisting of workers and potentially a dedicated listener. We further reduce latencies caused by disk I/O, placing the MariaDB data directory in a filesystem mapped to main memory.

We assign MariaDB to all 24 cores of CPU 1 and BenchBase to 23 cores of CPU 0. As ten connections are used, each thread processing queries can be scheduled on its own physical core, leaving more than 10 cores for background threads.

These configurations and (system) optimizations enable accurate and highly repeatable measurements.

***Redis.* Code Extensions.** For Redis (version 7.0.11), we injected one quiescence point in the single-threaded main event loop.

Patches. Redis is implemented in C, i.e. it is highly compatible for patch generation with Kpatch. From the development history of Redis versions 5.0.0–7.0.11 on GitHub, we extracted 529 patches.

Benchmarks. We extended the vendor benchmark framework redis-bench (part of the Redis project) to measure individual latencies. We use benchmarks consisting of only `SET` or `GET` operations.

6.1 Developer Perspective: Impact of Quiescence

We assume the perspective of the database developer, concerned about the safety of quiescence points and performance regressions.

[7] https://github.com/cmu-db/benchbase; git hash: `979b53b043`

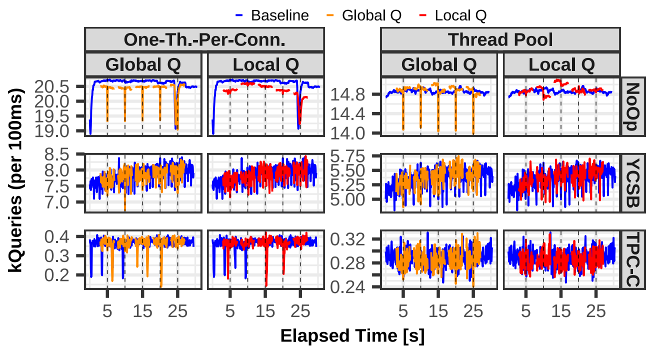


Figure 5: Query throughput over time for OLTP workloads, MariaDB without patch application ("baseline", blue) versus live patching in 5-second intervals in different setups.

6.1.1 ***OLTP Workloads.*** Figure 5 shows throughput over time aggregated over bins of 100 ms, running OLTP-benchmarks against MariaDB. We compare one-thread-per-connection (left) against thread pool (right). We patch the system (using patch ID #1) 5, 10, 15, 20, and 25 seconds into the benchmark (but each in a separate run) to catch the system in different states. We employ both global (left column; orange line) and local quiescence (right column; red line). We show the throughput over time 2 seconds before/after patch application (for a compact visualization and an easy comparison of the results). In each chart, the live patching run is superimposed onto the baseline run (blue line) for comparison.

The experiments for the remaining patches (patch IDs #2–#5) do not provide new insights. We refer to our artifacts and to the extended version of this article [20] for the corresponding charts.

Results. Obviously, throughput is highest for the lightweight NoOp benchmark and lower for more intense workloads. Throughput over time for live patching aligns closely with the baseline. Only a slight variation can be observed for the NoOp benchmark (e.g. about 100 queries per 100 ms difference). These marginal differences fall within the normal variations between individual runs.

For the NoOp benchmark, we can observe for all three configurations (baseline, global and local quiescence) of the one-thread-per-connection policy a brief drop in throughput at about 25 seconds. However, for global quiescence, we can additionally observe a short drop in throughput for both connection policies when performing a live patch. These drops cannot be observed for local quiescence; therefore, they can be attributed to threads blocking for global quiescence. Unlike with NoOp benchmark, which is lightweight and sensitive, throughput over time fluctuates with the YCSB and TPC-C benchmarks, masking any drops caused by global quiescence.

TPC-C has the lowest throughput for the one-thread-per-connection policy with about 350 queries per 100 ms, thus it takes on average about 0.35 ms for one thread to process one query. In global quiescence, this is also the average time that a thread has to wait for all other threads to reach their quiescence point (after which the patch is applied). Thus, live patching under an OLTP workload does not noticeably impact throughput.

One concern from the developer perspective is that of encouraging deadlocks. While we did do not encounter deadlocks in any run,

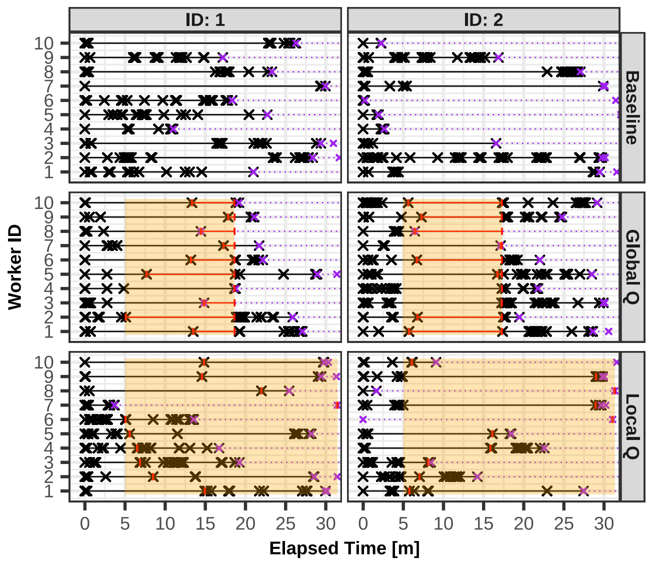


Figure 6: Fine-granular traces of the activities of 10 worker threads in MariaDB. Top row shows baseline. Yellow background highlights the synchronization time.

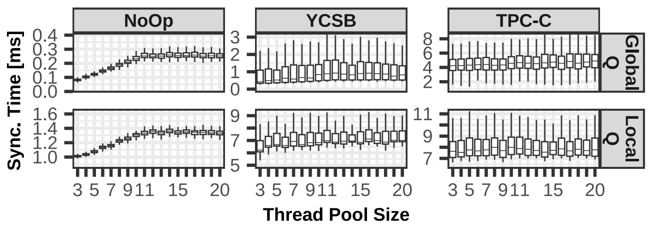


Figure 7: Boxplots of synchronization times, varying thread pool size and triggering patch application every 100 ms.

this is not the case for the earlier implementation [49] by Rommel et al. for the MariaDB thread pool policy, which does not employ the priority-based quiescence proposed by us. When we replicate our experiment using the YCSB and TPC-C benchmark with their implementation, every patch request inevitably causes a deadlock (in 100 out of 100 runs). This illustrates the intricacies of identifying suitable quiescence points and justifies our approach.

6.1.2 *OLAP Workload*.

Global quiescence is unproblematic with short-lived queries, as the threads will frequently encounter their quiescence points. But with long-running queries, threads synchronizing at the quiescence barrier may incur longer wait times. Therefore, we discuss our experiment with our OLAP workload and the one-thread-per-connection policy.

One-thread-per-connection. Figure 6 shows fine-grained traces of ten worker threads in MariaDB running the one-thread-per-connection policy. The topmost row shows baseline runs (no patches applied). Randomness in issuing queries in the benchmark harness and differences in query runtimes lead to unique traces.

Each line shows the activity of one given worker thread over time. The start and end of a query are marked with a cross and a connecting line. A query that does not complete within the 30 minute measurement phase is shown as a small purple cross, connected by a dotted purple line. The pale yellow rectangle frames the synchronization time, that is, the time until all threads have migrated to the new version. Red bars connected by a red line show the time a thread blocks at a quiescence point until it has migrated.

The second row shows traces of patch ID #1 and ID #2, applied with the global quiescence method. The third row shows traces of patch ID #1 and ID #2 applied, but now with the local quiescence method. Further traces are included in our artifacts and our extended version of this article [20].

Results. We first focus on patch ID #1 under global quiescence (Figure 6, left middle chart): Worker 2 reaches its quiescence point about 5 seconds after the patch request is issued. It blocks for about 13 minutes until worker 4 reaches its quiescence point (last thread to reach its barrier). Therefore, all workers that have reached their quiescence point before worker 4 remain idle and blocked. As expected, global quiescence incurs long wait times.

Local quiescence results in evidently higher concurrency (more crosses within the yellow-shaded area), as threads promptly migrate upon encountering a quiescence point. Yet, overall, it takes longer for all workers to complete their migration: As all worker threads keep working, they keep competing for resources (locks), which can delay the other threads in reaching their quiescence points. In both bottom charts, the last worker completes migration about one minute *after* the 30-minute benchmark window.

6.1.3 *Synchronization Time*.

To evaluate synchronization time for the thread pool policy, we perform an experiment using OLTP benchmarks and the source code version of patch ID #1 (patch ID #2–#5 show the same effects and the results are available in the artifacts and our extended version of this article [20]). For the duration of the 30-second benchmark phase, we trigger patch application every 100 ms, i.e. 300 patch requests per run. We measure synchronization time, but *without* applying a real patch. Patch application is highly patch-dependent (an effect which we explore below). For global quiescence, we measure the time until all threads reach their barrier and for local quiescence the duration of cloning the AS plus the time until all threads have migrated to the new AS generation. The experiment is conducted for a scale-out scenario, ranging the thread pool size from three to 20 (keep in mind that our benchmark utilizes ten parallel connections that are uniformly distributed among all thread groups in round-robin fashion).

The boxplots in Figure 7 show synchronization times, where outliers (data points outside the boundary of the whiskers; whiskers are based on the 1.5 IQR value) are not shown for better visualization. Columns specify the benchmark, while rows specify the quiescence method. The x-axis denotes thread pool size, and the y-axis the synchronization time in milliseconds (y-axes scaled individually).

As can be expected, the synchronization time is inversely correlated with throughput (see Figure 5): The higher the throughput, the lower the synchronization time, since quiescence points are passed more frequently. However, this is not the case for the OLAP workload visualized in Figure 6: The synchronization time for global quiescence is lower compared to local quiescence, even though throughput is lower. This advantage comes with the drawback of

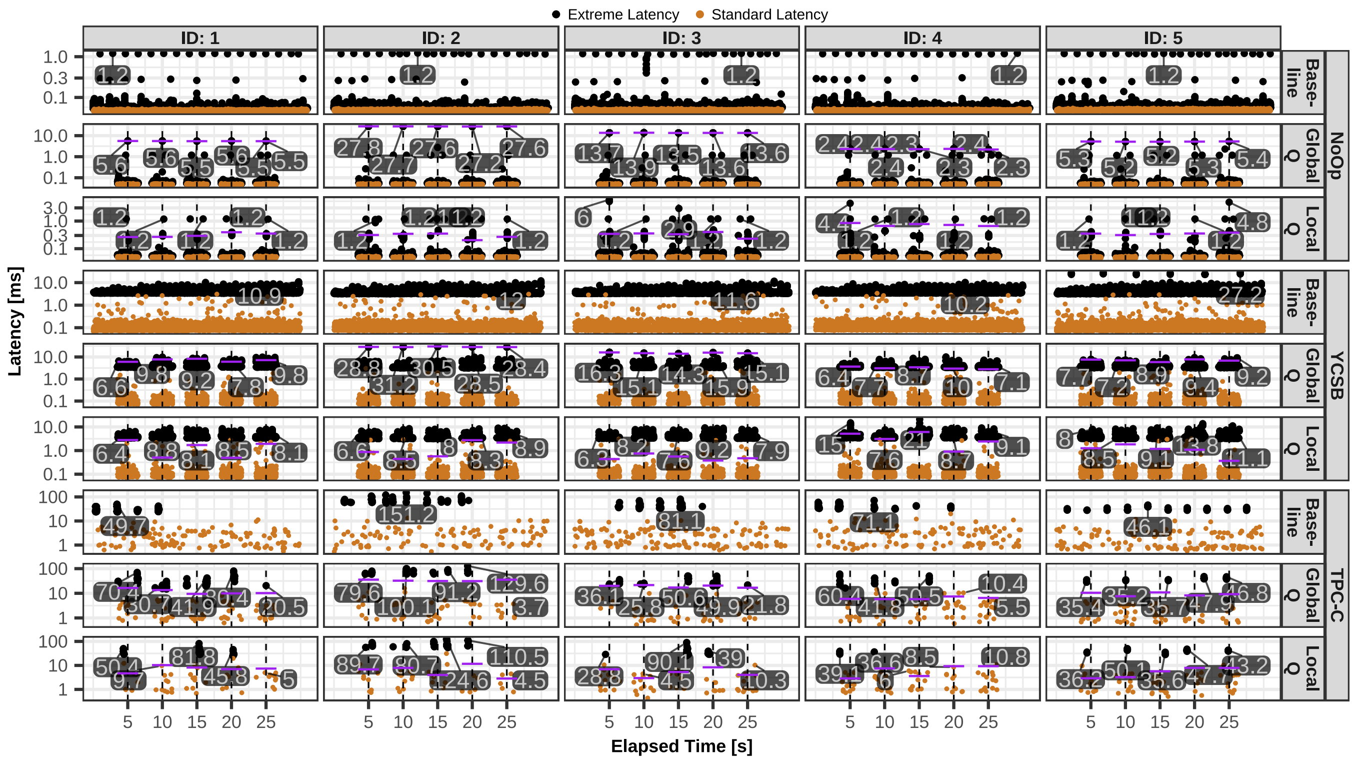


Figure 8: Latencies in live patching MariaDB under the one-thread-per-connection policy. Varying patches and setup.

threads blocking at their quiescence points, leading to less competition for locks, but also to a reduced degree of parallelism.

For the NoOp benchmark in Figure 7, the synchronization time increases with the number of thread groups, but this is noticeable only up to a thread pool size of ten. This indicates that synchronization time increases with the number of thread groups, but this effect is notable only when the thread groups are active (i.e., have an assigned connection). Inactive thread groups do not deteriorate the synchronization time. A similar trend is observed for global quiescence in the YCSB and TPC-C benchmarks, albeit to a lesser extent due to the generally higher synchronization time.

For this experiment, no deadlock appeared during any of the approximately 81,000 patch applications with global quiescence. This once again highlights – albeit empirically – the applicability of our priority-based quiescence approach for thread pools.

6.2 Client Perspective: Extreme Latencies

The experience of the database client is shaped by latencies in query processing, as extreme latencies can accumulate in distributed systems [13]. Figure 8 visualizes[8] query latencies in live patching MariaDB with the one-thread-per-connection policy. We compare five patches, different OLTP benchmarks, and quiescence methods.
Baseline. The top row shows a baseline run for the NoOp benchmark where no patches are applied. Along the horizontal axis, we show progress over time. The dots represent latencies measured within BenchBase and reflect the experience of the database client. Latencies beyond the 99.95th percentile are colored black (extreme values). The maximum latency is labeled. The standard latencies in orange are heavily sampled (down to 10%) to reduce overplotting.

Comparing the baseline runs (rows 1, 4 and 7), we confirm that the more intensive the workload, the higher the extreme latencies in the baseline runs. These charts not only underscore the robustness and repeatability of our experiments but also highlight the influence of code versions on performance. When examining charts for different patch IDs, a highly consistent latency pattern is apparent. However, occasional variations in extreme latencies arise (e.g., TPC-C baseline row) due to each MariaDB patch being associated with a specific code version, with an individual performance profile.
Live Patching. We focus on the 2nd and 3rd charts (counting from the top) in the left column, showing live patching of MariaDB under the NoOp benchmark for patch ID #1. We perform five isolated runs and issue a patch request at either 5, 10, 15, 20, or 25 seconds into the benchmark (to hit the system in different states). The charts show the time slices 1.5 seconds before/after patch application (for a compact visualization and an easy comparison of the results). By comparing the latencies at the time of patch application against the baseline run, we can observe the overhead of live patching. Moreover, the horizontal purple lines visualize the synchronization time for global quiescence and, for local quiescence, the longest duration from a quiescence point to migration completion.
Results. We focus on patch #1 and the NoOp benchmark. In the baseline run, we observe standard latencies of about 0.03 ms and

[8]For this sequence of charts, we adapted the visualization style and the scripts from a reproduction package for an earlier project [21].

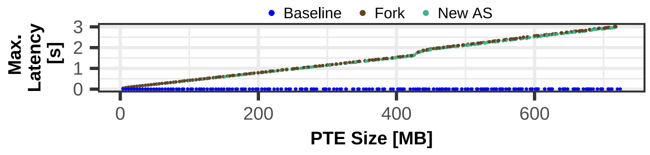


Figure 9: Impact of size of database state on max. query latencies, comparing address space cloning and forking in Redis.

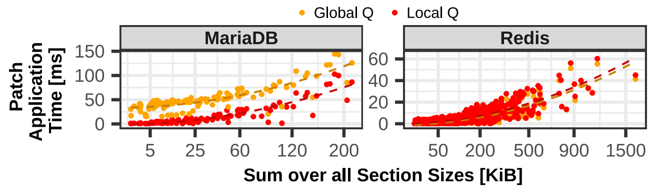


Figure 10: Impact of patch size on patch application time.

extreme latencies in the 1 ms range. With global quiescence, we observe one extreme latency of about 5.6 ms at patch application time. This corresponds to the time it takes for all threads to migrate (purple line) and can clearly be attributed to patch application. Note that these extreme latencies include the time until all threads have reached their barrier and the time required to load and apply the patch (to be explored in Section 6.3). For local quiescence, we only observe a slight increase in extreme latencies. The purple line shows the maximum time a thread needs to migrate, which is about 0.3 ms.

Let us also compare the behavior of different patches. The baseline runs are highly similar under the NoOp benchmark, However, different patches cause different extreme latencies. With global quiescence, the maximum latencies of patch #2 are at about 27.8 ms and for patch #3 at about 13.9 ms. This suggests that the time for loading and applying the patch is patch-specific. We explore this effect in Section 6.3. A similar behavior can be observed for YCSB, but not for TPC-C: This benchmark is more work-intensive, and the latency overhead is still within the standard range of TPC-C.

6.3 Provider Perspective: Predictable Overheads

For the database provider, it is crucial to be able to assess the factors influencing the overhead of live patching.

Size of Database State. Rommel et al. [48] carefully explored key impact factors from the perspective of operating systems research. They measured the overhead for address space cloning and switching (both operations in local quiescence) across six different user-space applications, each executed with a fixed configuration. Their experiment shows that creating an address space scales with its size, while migration is a constant-time operation.

For the database provider, the former is a concern, since DBMSs commonly have large memory states. To explore this further, we turn to Redis, a memory-based key-value store where we can easily control memory consumption. For example, we can issue SET operations with a data size of 400 KiB. The page size on our machine is 4096 B, so each SET results in the allocation of 100 pages (neglecting other internal data structures). As the page table stores one pointer for each page (the page table entry), and a pointer has 8 B, this results in 800 B of new page table entries (PTEs). The size of the page table is therefore a proxy metric for the number of page table entries (neglecting details of the internal page table tree structure) and correlates with main memory usage. Thus, SET operations inflate the page table and also serve as benchmark workload.

We create 200 instances of Redis with a total page table size of up to 818 MiB. Figure 9 visualizes the *maximum* query latencies in each baseline run (blue ticks). These are not affected by the size of the memory state. The (modified) patcher thread triggers Redis to clone an address space while Redis is under load. We capture the maximum query latency within a window of ±1 second (green ticks). For reference, the patcher thread triggers Redis to fork() instead of creating a new address space (brown ticks). The results for address cloning and forking are near-indistinguishable.

Results. The maximum latencies increase linearly with the size of the page table, i.e. the size of the database state. An administrator who knows the size of the allocated memory can predict the maximum latencies to be expected in live patching. However, even the delay in the range of seconds still outperforms a database restart, as it takes several minutes to restore 120 GB of data from shared-memory [22]. It is also important to note that cloning an address space only needs to copy 8 B (PTE) per 4 KiB of data (page).

AS cloning is implemented similarly to the fork() system call [48], which results in highly similar latencies. AS cloning also suffers from the process freeze of fork() [26, 47]: For copying the memory map structure, *all* threads are temporarily halted, so the latency affects *all* threads, not just the one initiating AS cloning.

Size of the Binary Patch. We measured the impact of the patch size on the patch application time for two systems. For MariaDB, we use a total of 117 patches, for Redis a total of 529 patches, all real-world patches extracted from GitHub. We only measure the patch application time from within the WfPatch user-space library.

Figure 10 plots the size of the binary patches (stating the sum of their section sizes) against the time it takes to actually apply the patch. We compare the global and local quiescence method. Recall that in the first case, the patch is applied directly to the address space (in place); in the second case, the patch is applied to a cloned address space. In this experiment, barrier wait times etc. are ignored. We put both databases under load (NoOp for MariaDB and GET for Redis), so that the threads can reach their quiescence points.

Figure 10 shows the relationship between the total sum of the size of all sections of the patch file (x-axis; please note square root scaling) and the patch application time, for global quiescence (orange) and local quiescence (red). A dashed regression line has been imposed for both quiescence methods. We observe that the time for patch application increases with the size of the patch. However, the latencies of MariaDB (one-thread-per-connection) are higher than those for Redis. For MariaDB, applying a patch with a total section size of about 200 KiB takes about 50 ms using local quiescence and 85 ms using global quiescence. For Redis, latencies are in the range of 7 ms. This reveals that patch application for MariaDB with local quiescence is faster than with global. This observation cannot be made for Redis.

Results. The duration of patch application depends on the size of the patch binary files. Moreover, for the multi-threaded MariaDB, patch application takes longer and the duration depends on the quiescence method used, which is not the case for Redis.

Table 2: Minimum and maximum latencies of live patching operations observed throughout the experiments.

	Reach Quiescence	Apply Patch	New AS
Min. / Max.	0.4 µs / 26 min	0.1 ms / 145 ms	1 ms / 3 s

7 DISCUSSION AND OUTLOOK

Our study is the first to evaluate the potential of live patching of multi-threaded DBMS from the perspective of *database systems research*. In the following, we summarize insights and challenges.

Main Insights. Our experiments show that live patching can indeed be a viable alternative to conventional means of updating, given that the patch is suitable. In our experiments, the quiescence points injected in the code for connection management did not cause any deadlocks.[9] This makes priority-based quiescence in thread pools one of our core contributions.

Our experiments show that the observed extreme latencies are in the milliseconds or lower second range. In comparison, a DBMS restart (1) loses all connections, (2) cannot create new connections or respond during the downtime and (3) takes several minutes when restoring the database state from shared-memory, or even hours to restore from disk (for 120 GB [22]).

Latency Breakdown. Our experiments examine the time taken by the individual live patching operations and the factors that influence it. Table 2 shows the lowest and highest latencies observed in our experiments. The overall latency of global quiescence comprises the maximum latency among individual threads reaching their quiescence point ("Reach Quiescence" column), which varies with the workload (Figure 6 and Figure 7), plus the subsequent loading and application of the patch ("Apply Patch" column), which is specific to the patch (Figure 10). With local quiescence, the total latency comprises (1) creating a new address space ("New AS" column), which is proportional to the size of the memory state (Figure 9). (2) The WfPatch thread switches to the new address space, and (3) applies the patch, both actions are executed in the background. (4) Eventually, threads reach their quiescence point, and (5) switch to the patched address space. Switching address spaces is a constant-time operation [48], in our experiments in the range of 4 µs – 2 ms. In summary, the workload, the size of the DBMS memory state, and the specific patch independently influence latencies.

Exploring Trade-Offs. For thread-local patches, we may choose between local or global quiescence. Our experiments show that there is no direct answer to the question of which method to prefer. Regardless of the workload, global quiescence displays lower synchronization times. This can help close software vulnerabilities quickly. However, blocking threads reduce the degree of concurrency until global quiescence is reached. In particular, this negatively affects OLAP workloads. In addition, there is a delay in loading and applying a patch, which depends on the patch size and also partly on the quiescence method. The duration of patch application is important not only for synchronization time, but also temporarily impose latency overheads.

[9]To *guarantee* the absence of deadlocks, specialized methods such as code analysis, model checking, etc., must be employed, which is beyond the scope of this paper.

Main memory databases are particularly affected, especially by AS cloning of WfPatch (with local quiescence). The larger the main memory state, the longer the entire process is frozen. This increases extreme latencies and also the synchronization time.

Challenges for Database DevOps. Integrating live patching into the database DevOps workflow raises several challenges. Live patching requires custom Linux kernels and libraries, and such major changes to the systems stack require extensive testing. Live patching even affects the way code changes are prepared when they are intended to be applied as a live patch. The Linux community observes best practices for carrying out code modifications [16] to ensure that patches can be generated seamlessly. Given a code change, DBMS vendors must carefully analyze its behavioral changes, possibly supported by program analysis tools. In [19], we further outline our vision of this new DevOps workflow.

Challenges in Tooling. From our own experience, debugging in the context of live patching is a serious challenge. Debuggers such as the GNU debugger (GDB) cannot be used out-of-the-box for applications having multiple address spaces. If the application crashes, the core dump cannot be analyzed using Linux on-board tools. Over time, we can expect the tooling ecosystem to evolve with live patching in user-space applications becoming more common.

Challenges in Patching Database Clusters. So far, we have evaluated live patching for a single-instance, multi-threaded DBMS. In another line of work, we address live patching of *distributed* in-memory key value stores [19] and systematically explore the swift and reliable dissemination of patches in the cluster. Our results demonstrate that live patching outperforms conventional patching via rolling updates, specifically maintaining stable throughput, avoiding latency spikes, and only marginally increasing network consumption during the distribution of patches across the cluster. We refer to [19] for a detailed discussion of our findings and the remaining challenges in the context of distributed DBMSs.

Overall, having opened the field of research on live patching of databases, and having conducted our experiments and analyses, we see strong potential for follow-up research and real-world impact.

ACKNOWLEDGMENTS

The authors thank Ralf Ramsauer (OTH Regensburg), Christian Dietrich (Hamburg University of Technology) and Florian Rommel (Leibniz University Hannover) for technical support with WfPatch at early stages of this research, Stefan Klessinger (University of Passau) and Mojtaba Zali (University of Passau) for their support with crawling patches, and Julia Fruth for help with proofreading.

REFERENCES

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak R. Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. *Proc. VLDB Endow.* 6, 11 (2013), 1057–1067. https://doi.org/10.14778/2536222.2536231

[2] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security. In *Proc. USENIX Security Symposium.*

[3] Amazon. 2018. *Best Practices for Upgrading Amazon RDS for MySQL and Amazon RDS for MariaDB.* Retrieved October 23, 2023 from https://aws.amazon.com/blogs/database/best-practices-for-upgrading-amazon-rds-for-mysql-and-amazon-rds-for-mariadb/

[4] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: automatic rebootless kernel updates. In *Proc. EuroSys.* 187–198. https://doi.org/10.1145/1519065.1519085

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. USENIX*. 49–60.

[6] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. SIGMOD*. 265–276. https://doi.org/10.1145/1989323.1989352

[7] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. 2007. POLUS: A POwerful Live Updating System. In *Proc. ICSE*. 271–281. https://doi.org/10.1109/ICSE.2007.65

[8] Codership. [n.d.]. *Upgrading Galera Cluster.* Retrieved October 23, 2023 from https://galeracluster.com/library/documentation/upgrading.html

[9] Codership. 2013. *Minimizing downtime and maximizing elasticity with Galera Cluster for MySQL.* Retrieved October 23, 2023 from https://galeracluster.com/wp-content/uploads/2013/10/Minimizing-downtime-and-maximizing-elasticity-with-Galera-Cluster-for-MySQL.pdf

[10] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proc. DBTest*. 1–6. https://doi.org/10.1145/1988842.1988850

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proc. SoCC*. 143–154. https://doi.org/10.1145/1807128.1807152

[12] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proc. SIGMOD*. 215–226. https://doi.org/10.1145/2882903.2903741

[13] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. https://doi.org/10.1145/2408776.2408794

[14] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288. https://doi.org/10.14778/2732240.2732246

[15] dynup/kpatch. 2024. kpatch: dynamic kernel patching. Retrieved January 19, 2024 from https://github.com/dynup/kpatch

[16] dynup/kpatch. 2024. kpatch Patch Author Guide. Retrieved January 19, 2024 from https://github.com/dynup/kpatch/blob/master/doc/patch-author-guide.md

[17] EDB. [n.d.]. *PostgreSQL BDR (Bi-Directional Replication).* Retrieved October 23, 2023 from https://www.enterprisedb.com/docs/pgd/4/bdr/

[18] Michael Fruth. 2022. Live Patching Database Management Systems. In *Proc. SIGMOD*. 2524–2526. https://doi.org/10.1145/3514221.3520253 ACM SIGMOD Student Research Competition 2022.

[19] Michael Fruth and Stefanie Scherzinger. 2024. Live Patching for Distributed In-Memory Key-Value Stores. In *Proc. SIGMOD*. https://doi.org/10.1145/3698816

[20] Michael Fruth and Stefanie Scherzinger. 2024. The Case for DBMS Live Patching [Extended Version]. https://doi.org/10.48550/arXiv.2410.09925 arXiv:2410.09925

[21] Michael Fruth, Stefanie Scherzinger, Wolfgang Mauerer, and Ralf Ramsauer. 2021. Tell-Tale Tail Latencies: Pitfalls and Perils in Database Benchmarking. In *Proc. TPCTC*, Vol. 13169. 119–134. https://doi.org/10.1007/978-3-030-94437-7_8

[22] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet L. Wiener. 2014. Fast database restarts at facebook. In *Proc. SIGMOD*. 541–549. https://doi.org/10.1145/2588555.2595642

[23] Deepak Gupta, Pankaj Jalote, and Gautam Barua. 1996. A Formal Framework for On-line Software Version Change. *IEEE Trans. Software Eng.* 22, 2 (1996), 120–131. https://doi.org/10.1109/32.485222

[24] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. 2012. Kitsune: efficient, general-purpose dynamic software updating for C. In *Proc. OOPSLA*. 249–264. https://doi.org/10.1145/2384616.2384635

[25] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. 2011. State transfer for clear and efficient runtime updates. In *Proc. ICDE*. 179–184. https://doi.org/10.1109/ICDEW.2011.5767632

[26] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[27] The kernel development community. [n.d.]. Livepatch - The Linux Kernel documentation. Retrieved October 23, 2023 from https://docs.kernel.org/livepatch/livepatch.html

[28] Rajender Kumar. 2021. *IBM AIX 7.2 Live Kernel Update for a reboot-free world!* Retrieved October 23, 2023 from https://www.ibm.com/support/pages/ibm-aix-72-live-kernel-update-reboot-free-world

[29] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25. https://doi.org/10.1145/3588687

[30] Kristis Makris and Rida A. Bazzi. 2009. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *Proc. USENIX*.

[31] MariaDB. [n.d.]. *Changing a Replica to Become the Primary.* Retrieved October 23, 2023 from https://mariadb.com/kb/en/changing-a-replica-to-become-the-primary/

[32] MariaDB. [n.d.]. *Replication Overview.* Retrieved October 23, 2023 from https://mariadb.com/kb/en/replication-overview/

[33] MariaDB. [n.d.]. *What is MariaDB Galera Cluster?* Retrieved October 23, 2023 from https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/

[34] Microsoft. 2023. *Hotpatch for virtual machines.* Retrieved October 23, 2023 from https://learn.microsoft.com/en-us/windows-server/get-started/hotpatch

[35] Microsoft. 2023. *Upgrade a failover cluster instance.* Retrieved October 23, 2023 from https://learn.microsoft.com/en-us/sql/sql-server/failover-clusters/windows/upgrade-a-sql-server-failover-cluster-instance?view=sql-server-ver16

[36] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. 2013. RemusDB: transparent high availability for database systems. *VLDB J.* 22, 1 (2013), 29–45. https://doi.org/10.1007/s00778-012-0294-6

[37] MySQL. [n.d.]. *MySQL Cluster CGE.* Retrieved October 23, 2023 from https://www.mysql.com/products/cluster/

[38] Iulian Neamtiu, Michael W. Hicks, Gareth Paul Stoyle, and Manuel Oriol. 2006. Practical dynamic software updating for C. In *Proc. SIGPLAN*. 72–83. https://doi.org/10.1145/1133981.1133991

[39] 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 *Proc. NSDI*. 385–398.

[40] Hans Olav Norheim. 2019. *Hot Patching SQL Server Engine in Azure SQL Database.* Retrieved October 23, 2023 from https://techcommunity.microsoft.com/t5/azure-sql-blog/hot-patching-sql-server-engine-in-azure-sql-database/ba-p/849700

[41] Pu Pang, Gang Deng, Kaihao Bai, Quan Chen, Shixuan Sun, Bo Liu, Yu Xu, Hongbo Yao, Zhengheng Wang, Xiyu Wang, Zheng Liu, Zhuo Song, Yong Yang, Tao Ma, and Minyi Guo. 2023. Async-fork: Mitigating Query Latency Spikes Incurred by the Fork-based Snapshot Mechanism from the OS Level. *Proc. VLDB Endow.* 16, 5 (2023), 1033–1045. https://doi.org/10.14778/3579075.3579079

[42] Josh Poimboeuf. 2014. *Introducing kpatch: Dynamic Kernel Patching.* Retrieved October 23, 2023 from https://www.redhat.com/de/blog/introducing-kpatch-dynamic-kernel-patching

[43] PostgreSQL. [n.d.]. *Replication.* Retrieved October 23, 2023 from https://www.postgresql.org/docs/current/runtime-config-replication.html

[44] PostgreSQL. [n.d.]. *Upgrading a PostgreSQL Cluster.* Retrieved October 23, 2023 from https://www.postgresql.org/docs/current/upgrading.html

[45] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. 2013. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proc. ADMS*. 36–45.

[46] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10, 2 (2016), 37–48. https://doi.org/10.14778/3015274.3015275

[47] Redis. [n.d.]. *Diagnosing latency issues.* Retrieved October 23, 2023 from https://redis.io/docs/management/optimization/latency/

[48] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. 2020. From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes. In *Proc. OSDI*. 651–666.

[49] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. 2020. Reproduction Package for From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes. Retrieved October 23, 2023 from https://www.sra.uni-hannover.de/Publications/2020/WfPatch/index.html

[50] Florian Rommel, Christian Dietrich, Andreas Ziegler, Illia Ostapyshyn, and Daniel Lohmann. 2023. Thread-Level Attack-Surface Reduction. In *Proc. LCTES*. 64–75. https://doi.org/10.1145/3589610.3596281

[51] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *Proc. VLDB Endow.* 9, 10 (2016), 768–779. https://doi.org/10.14778/2977797.2977803

[52] SUSE. 2023. libpulp. Retrieved October 23, 2023 from https://github.com/SUSE/libpulp

[53] SUSE. 2023. *Live Kernel Patching Using kGraft.* Retrieved October 23, 2023 from https://documentation.suse.com/sles/12-SP4/html/SLES-kgraft/index.html

[54] Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. 2023. MELF: Multivariant Executables for a Heterogeneous World. In *Proc. USENIX*. 257–273.

[55] Transaction Processing Council. 2010. TPC-C Benchmark (Revision 5.11). Retrieved October 23, 2023 from http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[56] Nico Weichbrodt, Joshua Heinemann, Lennart Almstedt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2021. sgx-dl: dynamic loading and hot-patching for secure applications: experience paper. In *Proc. Middleware*. 91–103. https://doi.org/10.1145/3464298.3476134

[57] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proc. OSDI*. 465–477.