Automating Live Update for Generic Server Programs

Cristiano Giuffrida, *Member, IEEE,* Călin Iorgulescu,
Giordano Tamburrelli and Andrew S. Tanenbaum, *Fellow, IEEE,*

Abstract—The pressing demand to deploy software updates without stopping running programs has fostered much research on live update systems in the past decades. Prior solutions, however, either make strong assumptions on the nature of the update or require extensive and error-prone manual effort, factors which discourage the adoption of live update.
This paper presents *Mutable Checkpoint-Restart* (*MCR*), a new live update solution for generic (multiprocess and multithreaded) server programs written in C. Compared to prior solutions, MCR can support arbitrary software updates and automate most of the common live update operations. The key idea is to allow the running version to safely reach a quiescent state and then allow the new version to restart as similarly to a fresh program initialization as possible, relying on existing code paths to automatically restore the old program threads and reinitialize a relevant portion of the program data structures. To transfer the remaining data structures, MCR relies on a combination of precise and conservative garbage collection techniques to trace all the global pointers and apply the required state transformations on the fly. Experimental results on popular server programs (*Apache httpd*, *nginx*, *OpenSSH* and *vsftpd*) confirm that our techniques can effectively automate problems previously deemed difficult at the cost of negligible performance overhead (2% on average) and moderate memory overhead (3.9x on average, without optimizations).

Index Terms—Live update, DSU, Checkpoint-Restart, Quiescence detection, Record-Replay, Garbage collection

1 Introduction

Live update—also commonly known as *dynamic software updating* [1]—has gained momentum as a solution to the *update-without-downtime* problem, that is, deploying software updates without stopping running programs or disrupt their state. Compared to the most common alternative—that is, *rolling upgrades* [2]—live update systems require no redundant hardware and can automatically preserve program state across versions. Ksplice [3] is perhaps the best known live update success story. According to its website, Ksplice has already been used to deploy more than 2 million live updates on over 100,000 productions systems at more than 700 companies.

Despite decades of research in the area—with the first paper on the subject dating back to 1976 [4]—existing live update systems still have important limitations. For example, *in-place* live update solutions [1], [3], [5], [6], [7] can transparently replace individual functions in a running program, but are inherently limited in the types of updates they can support without significant manual effort. Ksplice, for instance, is explicitly tailored to small and simple security patches [8]. Conversely, existing *whole-program* live update solutions [9], [10] can efficiently support several classes of updates, but require a nontrivial annotation effort which significantly increases the maintenance burden and ultimately discourages adoption of live update.

This paper presents *Mutable Checkpoint-Restart* (*MCR*), a new live update solution for generic server programs written in C. Building on kernel support for emerging user-space checkpoint-restart techniques [11], MCR (i) checkpoints the running version—safely allowing its execution to reach a *quiescent state* [12]—(ii) restarts the new version—reinitializing it from scratch in a controlled way—(iii) restores the checkpointed state in the new version—automatically transferring the necessary state (e.g., open connections) from the old version. This is similar, in spirit, to a classic checkpoint-restart model [11], but the mutability between versions yields a *whole-program* live update strategy with support for arbitrarily complex software updates.

To minimize the annotation effort involved, MCR relies on three key observations. First, while safely allowing an arbitrary set of threads to reach a quiescent state is extremely challenging with no assumptions on the program internals [5], real-world server programs generally exhibit a well-defined quiescent behavior, with a limited number of long-running tasks receiving, processing, and dispatching server events. Building on this observation, MCR relies on *profile-guided quiescence* to record the behavior of a server program offline and automatically drive all its threads into a quiescent state at runtime. In

- *C. Giuffrida, G. Tamburrelli, and A.S. Tanenbaum are with the Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081HV Amsterdam, The Netherlands.*
  E-mail: `{giuffrida,g.tamburrelli,ast}@cs.vu.nl`
- *Călin Iorgulescu is with École Polytechnique Fédérale de Lausanne, Lausanne 1015, Switzerland.*
  E-mail: `calin.iorgulescu@epfl.ch`

particular, the profiling information gathered allows MCR to safely quiesce all the program threads with *fast convergence* and in a *deadlock-free* fashion. Second, while transferring the *entire* execution state between different program versions is a notoriously hard problem [13] and generally requires significant manual effort [9], [10], this is not necessary for real-world server programs. Such programs typically initialize most of their state at startup and introduce only relatively small changes during their regular execution. Building on this observation, MCR relies on *mutable replay* techniques [14] to allow the new version to start up as similarly to a fresh program initialization as possible. Piggybacking on existing code paths allows for the seamless reinitialization of a relevant portion of the updated state. This *mutable reinitialization* strategy allows code in the new version to automatically restore updated program threads and many data structures—possibly subject to very complex changes between versions—with little annotation effort. Third, when transferring the data structures that cannot be automatically restored by *mutable reinitialization*, precise knowledge of data types in memory—which generally imposes a nontrivial annotation effort at full coverage [9], [10]—is only necessary for updated data structures that need to be type-transformed between versions. Building on this observation, MCR relies on a combination of *precise* [15] and *conservative* [16], [17] garbage collection (GC) techniques to trace data structures and transfer them between versions even with partial type information. This *mutable tracing* strategy can drastically reduce the number of developer-maintained annotations. These are only required when data structures with ambiguous type information—and thus normally traced conservatively—are changed by the update—and thus require precise tracing to unambiguously apply type transformations.

Summarizing, we make the following contributions:

- We present *profile-guided quiescence*, a technique which allows all the program threads to automatically and safely block in a known quiescent state using dedicated information gathered during an offline profiling phase.
- We present *mutable reinitialization*, a technique which record-replays startup operations between different program versions and exploits existing code paths to automatically reinitialize the new program version, its threads, and a relevant portion of its global data structures.
- We present *mutable tracing*, a technique which transfers the remaining data structures between versions using precise (when possible) and conservative (otherwise) GC-style tracing strategies.
- We demonstrate the effectiveness of our techniques in *Mutable Checkpoint-Restart* (*MCR*), a new live update solution for generic server programs written in C. We present its implementation on Linux and evaluate it on 4 popular server programs, showing that MCR yields: (i) low engineering effort to support even complex updates (334 annotation LOC in total to prepare our programs for MCR), (ii) realistic update times (< 1 s); (iii) negligible performance overhead in the default configuration (2% on average); (iv) moderate memory overhead (3.9x on average, without optimizations).

The remainder of the paper is organized as follows. Section 2 illustrates prior work and techniques in the area of live update. Section 3 provides a high-level overview of the proposed solution. Sections 4, 5, and 6 provide a detailed description of the three main techniques that constitute the MCR approach: *(i)* profile-guided quiescence, *(ii)* mutable reinitialization, and *(iii)* mutable tracing, respectively. Section 7 provides an in-depth discussion of the relevant implementation details, while Section 9 presents experimental results to validate the proposed solution. Finally, Section 10 summarizes the findings of the paper.

2 Related Work

In this paper, we focus on *local* live update solutions for operating systems and long-running C programs, referring the reader to [2], [25], [26], [27] for live update for distributed systems.

This section illustrates related work and is organized into three distinct subsections that map to the main steps involved in live update solutions (i.e., *quiescence detection*, *control migration*, and *state transfer*). Table 1 presents an overview of the most relevant related approaches classified by category and adopted techniques.

2.1 Quiescence detection

Similar to prior work in the area, MCR relies on *quiescence* [12] as a way to restrict the number of possible program states at checkpointing time and ensure that live updates are only deployed in safe updates states (e.g., all the program threads blocked waiting for socket events). In other words, quiescence is used to guarantee update safety. Some approaches [24] relax this constraint, but then automatically remapping all the possible program states between versions—or simply allowing mixed-version execution [7], [18], [19]—becomes quickly intractable without extensive developer annotations. Quiescence detection algorithms proposed in prior work operate at the level of individual functions [3], [6], [20], [21] or generic events [1], [5], [10], [22], [23], [28]. The former approach is known for its weak consistency guarantees [10], [29] and typically relies on passive *stack inspection* [3], [6], [20], [21] that cannot guarantee convergence in bounded time [18], [24].

TABLE 1: Related Work: Overview

Category	**Techniques**				**Our Approach (MCR)**
Quiescence	Mixed execution: [7], [18], [19]	Individual function quiescence: [3], [6], [20], [21]	Design-induced quiescence: [10], [22], [23]	Explicit per-thread quiescence: [1], [5], [9], [24]	*Profile-guided quiescence*
	Strengths				
	No quiescence needed	Ease of implementation	Native support	Ease of implementation	Guaranteed time-bounded convergence
	Weaknesses				
	Extensive annotations required	Weak consistency guarantees	Not suitable for existing servers	No time-bounded convergence guarantees	Offline profiling required
Control Migration	In-place updates: [1], [3], [5], [6], [7], [18], [19], [22]	Design-induced control migration: [10], [23]	Stack reconstruction: [24]	Explicit control annotations: [9]	*Mutable reinitialization*
	Strengths				
	No control migration needed	Native support	Immediate updates	Ease of implementation	Automatically reuses existing code paths
	Weaknesses				
	Inhibit certain updates	Not suitable for existing servers	Manual call stack mapping required	Migration delegated to developer	May require annotations[1]
State Transfer	In-place updates: [3], [6], [7], [18], [19], [22]	Whole-program updates: [21], [23], [24]	Type transformers: [1], [5]	Precise GC-style tracing: [9], [10]	*Mutable Reinitialization and Mutable Tracing*
	Strengths				
	No state transfer overhead	No restrictions on state changes	Data structure changes handled automatically	Data structure changes handled automatically	Automated through hybrid GC techniques
	Weaknesses				
	State transfer delegated to developer	State transfer delegated to developer	No support for pointer transformations	Global pointer annotations required	May require annotations[1]

The latter approach relies on either update-friendly system design [10], [22], [23]—rarely an option for existing server programs written in C—or explicit per-thread *update points* [1], [5], [9], [24]—typically annotated at the top of per-thread long-running loops. Two update-point-based quiescence detection strategies are dominant: *free riding* [5], [24]—that is, allow program threads to run until they all happen to reach a valid update point at the same time–and *barrier synchronization* [9], [30]—that is, immediately block each thread at the next valid update point. The first strategy cannot guarantee convergence in bounded time. To mitigate this problem, prior solutions suggest expanding the number of update points using static analysis [5] or adopting per-function update points [24]. Both solutions can introduce substantial overhead, yet they still fail to guarantee convergence. The second strategy, on the other hand, offers better convergence guarantees but ignores interthread dependencies making it deadlock prone [5]. In addition, all the prior update-point-based strategies require abruptly interrupting the in-progress blocking calls, which would otherwise delay quiescence indefinitely. To address this problem, prior solutions suggest a combination of annotations and either signals [9] or file descriptor injection [24]. The former strategy is more general, but inherently race prone and potentially disruptive for the running program. MCR, in contrast, relies on *profile-guided quiescence* to safely and automatically quiesce all the program threads using offline profiling information. Unlike prior solutions, our quiescence detection strategy is designed to eliminate code annotations and provide efficient, race-free, and deadlock-free quiescence in bounded time. This is possible by controlling external and internal events individually processed by server programs in bounded time.

2.2 Control migration

Similar to prior work in the area, MCR relies on *control migration* [9] as a way to restore all the updated program threads after restart. Prior in-place live update models [1], [3], [5], [6], [7], [18], [19], [22]—which *patch* the existing program state in place to

1. See Section 8.

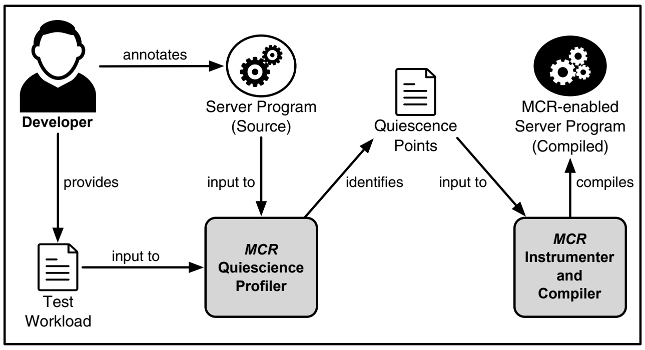
adapt it to the new version—however, provide no support for control migration. Instead, they implicitly forbid particular types of updates: Ksplice [3], for example, cannot easily support a simple update to a global flag that changes the conditions under which kernel threads enter a particular fast path. Failure to remap the latter may, for instance, introduce silent data corruption or synchronization issues such as deadlocks. Prior whole-program live update models, in turn, implement control migration using system design [10], [23], *stack reconstruction* [24], or annotations [9]. The first option is overly restrictive for many C programs. The second option exposes the developer to the heroic effort of remapping *all* the possible thread call stacks across versions. The last option, finally, reduces the effort by encouraging existing code path reuse, but still delegates control migration completely to the developer. MCR, in contrast, relies on *mutable reinitialization* to automatically reuse existing startup code paths in the new version and restore the program threads in the quiescent state equivalent to the one in the old version. Since server programs tend to naturally reach quiescence at startup, this strategy can drastically reduce the annotation effort required to complete control migration.

2.3 State transfer

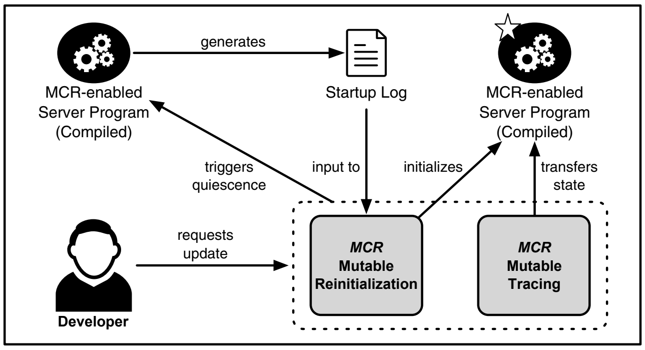
MCR relies on *state transfer* [21] as a way to remap the program state between versions (and to apply the necessary data structure transformations) after restart. Existing in-place live update solutions, however, either delegate state transfer entirely to the developer [3], [6], [7], [18], [19], [22] or provide simple type transformers with no support for pointer transformations [1], [5]. Such restrictions are inherent to the in-place live update model. Prior whole-program solutions, in turn, either delegate state mapping functions to the developer [23], [24] or attempt to reconstruct the state in the new version using *precise* GC-style tracing [9], [10]. The latter, however, requires a nontrivial annotation effort to identify all the global program pointers correctly. MCR, in contrast, relies on *mutable reinitialization* to allow existing startup code paths to seamlessly reinitialize a relevant portion of the program state, and on *mutable tracing* to automatically transfer the remaining portions between versions using *hybrid* GC techniques. The latter encourages annotationless semantics by tolerating partial pointer information and gracefully handling uninstrumented shared libraries and custom memory allocation schemes.

3 MCR Overview

The MCR approach consists of several steps as explained hereafter. The first two steps occur at build time (see Figure 1a) and produce a MCR-enabled version of a server program of interest. The following



(a) Build time



(b) Live update time

Fig. 1: MCR Overview

two steps occur instead at run time (see Figure 1b) and enable the live update process upon request.

1. *Program annotation and profiling.* To build a MCR-enabled version of a server program developers may need to annotate its source code (if necessary, as explained later) to allow the *MCR Quiescence Profiler* to run the annotated server under a given test workload. This preliminary step helps the developer identify the *quiescent points* in the program, later used by our instrumentation. This is a relatively infrequent operation which should only be repeated when the quiescent behavior of the program changes—we envision programmers simply integrating quiescence profiling as part of their regression test suite.

2. *Instrumentation and compilation.* Building a MCR-enabled version of the program requires only setting standard compiler flags which instruct the toolchain to link the code against the MCR static library and to enable the MCR link-time pass implemented in LLVM [31]. The latter instruments the profiled quiescent points for the benefit of our *profile-guided quiescence* strategy as explained later in Section 4.

3. *Startup recording and run-time monitoring.* During program startup, MCR records all the external operations (i.e., system calls) performed by the program in a *startup log*. This is later used by *mutable reinitialization* in the new version. After startup, MCR also efficiently monitors changes to existing data structures

```
1  /* Auxiliary data structures. */
2  char b[8];
3  typedef struct list_s {
4      int value;
5      struct list_s *next;
6  } l_t; l_t list;
7
8  /* Startup configuration. */
9  struct conf_s *conf;
10
11 /* Server implementation. */
12 int main() {
13    server_init(&conf); // startup
14    while(1) {          // main loop
15        void *e = server_get_event();
16        server_handle_event(e, conf, b, &list);
17    }
18    return 0;
19 }
20
21 /* MCR annotations. */
22 MCR_ADD_OBJ_HANDLER(b, custom_b_handler);
23 MCR_ADD_REINIT_HANDLER(custom_reinit_handler);
24 MCR_ADD_QUIESC_HANDLER(custom_quiesc_handler);
```

Listing 1: A sample MCR-enabled server program.

and marks those modified/created after startup as *dirty* in its internal metadata. This is later used by *mutable tracing* in the new version.

4. *Live update.* When an update is available, the user can signal the running version—using our `mcr-ctl` tool—to request a live update. In response, MCR first relies on our *profile-guided* quiescence to allow all the program threads to reach a *checkpoint* in a quiescent state. Next, it allows our *mutable reinitialization* to start up the new version from scratch, replay the necessary operations from the old startup log to prevent re-execution errors (e.g., attempt to rebind to port 80), restore the program threads, and reinitialize all the startup-time data structures and in-kernel state (e.g., open files, sockets, etc.). When startup in the new version completes, MCR allows our *mutable tracing* strategy to transfer the remaining (*dirty*) data structures from the old version to the new version. At the end of the process, MCR allows the new version to *restart* execution and terminates the old version.

Failure to complete the restart phase simply causes the new version to terminate and the old version to resume execution from the checkpoint. Since the entire MCR process is guaranteed to be atomic, the restart phase prevented from modifying the existing environment by mutable reinitialization, and the two program versions are completely isolated from one another, the failure is never exposed to the clients or propagated back to the old version by construction. This design yields a reversible live update strategy which can safely and automatically rollback all the failed update attempts—in contrast to prior user-level solutions for generic C programs [1], [5], [6], [7], [9], [24].

3.1 A simplified example

Listing 1 exemplifies a MCR-enabled server program with a simple but typical (event-driven) server structure. The program begins executing on line 13, with the entire startup code enclosed in the `server_init` function. Such function performs the necessary startup operations (e.g., socket creation) and also initializes the `conf` data structure containing the startup configuration (line 9) from persistent storage. After startup, the execution is confined in the long-running main loop on line 14, which, in each iteration, simply waits for a new event (e.g., a new connection) from the client and handles the event accordingly (e.g., sending back a welcome message). The function `server_get_event` (invoked on line 15) contains a natural *quiescent point* for the server program, given that execution may block waiting for events for an extended period of time with minimal in-flight state [9]. The function `server_handle_event` (invoked on line 16), in turn, handles each event in a timely fashion, possibly *reading* from the `conf` data structure and *writing* into the other *auxiliary* data structures (line 1).

During startup, MCR records all the operations performed by `server_init` in the startup log, until the program enters the main loop and *profile-guided quiescence* detects its first quiescent state in `server_get_event`. After startup, MCR detects changes to the auxiliary data structures and marks them as *dirty*. When live update is requested, *profile-guided quiescence* induces the main (and only) program thread to safely quiesce in `server_get_event`. Subsequently, MCR allows the new program version to independently start up. When the new version initializes (i.e., `server_init`), *mutable reinitialization* replays all the necessary operations from the recorded startup log. Simultaneously, *profile-guided quiescence* induces the new version to naturally reach its first quiescent state in `server_get_event` and block. With both versions now in an equivalent quiescent state, *mutable tracing* transfers all the *dirty auxiliary* data structures from the old version to the new version, omitting the (non-dirty) `conf` data structure automatically reinitialized by the startup code.

While MCR can, in principle, handle this simple scenario in a fully automated way, annotations (line 21) can be specified by the developer to handle more complex server structures and updates. For example, the `MCR_ADD_OBJ_HANDLER` state annotation at the bottom can help MCR identify *hidden* data structures and pointers stored in the `b` buffer (see example in Figure 4). The annotation takes as parameters the pointer to a data structure that MCR might have trouble identifying correctly and a pointer to a custom callback function. During live update, the annotation is called when MCR tries to infer the type of `b` and returns the missing type information (in a predefined format). *Mutable tracing* can normally handle these cases automatically, but cannot alone apply changes to such data structures when required by the update. Further, similar to prior solutions, state annotations are necessary to handle complex updates operating semantic changes to data structures or to external state (e.g., config files on persistent storage) [9], [32]. The `MCR_ADD_REINIT_HANDLER` annotation, in turn, can help *mutable reinitialization* replay startup operations when their semantics changes between versions or restore a quiescent state in the old version not automatically recreated at startup by the new version. For example, this is the case with servers that dynamically spawn threads/processes with long-lived *quiescent points* after startup. The `MCR_ADD_QUIESC_HANDLER` annotation, finally, can help *profile-guided quiescence* avoid undesirable quiescent states (e.g., those that are particularly challenging to restore in the new version) by invalidating particular per-thread quiescent points—that is, allowing blocking behavior for an extended period of time or gracefully returning an application-handled error code to the program. Each of the latter two annotations accept as parameter a pointer to a callback function invoked at a particular point in time (i.e., after reinitialization has finished and during quiescence detection, respectively).

4 Profile-guided Quiescence

Profile-guided quiescence, a crucial component of the MCR approach, has been conceived from two simple observations. First, the problem of transparently synchronizing multiple threads in bounded time and in a deadlock-free fashion is undecidable (i.e., easily reducible to the halting problem) without extra information on thread behavior. Second, every long-running program has a number of natural per-thread *execution-stalling points* [33] (i.e., program points in which thread execution may stall for an unbounded period of time), which are obvious choices to identify a globally quiescent state.

The key idea behind profile-guided quiescence is to profile the program at runtime and automatically identify *quiescent points* [29]—which specify program locations to safely block long-lived program threads and converge to a globally quiescent state—from all the stalling points observed.

We note a number of interesting stalling point properties in server programs. First, they always originate from long-lived blocking calls (e.g., `accept`) with well-known semantics. This allows us to automatically gather fine-grained information on a stalling thread and carefully control its behavior. Second, stalling points are often found at the top of long-running loops, which prior work has already largely recognized as ideal update points [1], [5], [9]. Third, even when stalling points are deeper in the call stack, fine-grained control over them is clearly crucial to reach quiescence, a common problem in prior work [9], [24].

The following two subsections illustrate in detail how MCR identifies quiescent points and how it exploits them through a dedicated protocol to identify quiescent states of a server program.

4.1 Quiescent point identification

To identify stalling points and the corresponding long-lived loops, our profiler relies on standard profiling techniques. In particular, MCR leverages *static instrumentation* to track all the threads (and processes) in the program and to intercept all the function calls, library calls, and loop entries/exits at runtime. The MCR Quiescence Profiler thus exploits *dynamic analysis* to identify all the classes of threads with the same stalling points. To produce the necessary profiling data (i.e., the dynamic traces), the profiler requires a test workload able to drive the program into all the potential execution-stalling states (e.g., a thread blocked on an idle connection, large file transfer, etc.) that must be accepted as legal quiescent states at live update time. In our experience, this workload is typically domain-specific—can be reused across several programs of the same class—and often simple to extrapolate from existing regression test suites. Even for very complex programs that may exhibit several possible stalling states, we expect this approach to be more intuitive, less error-prone, and more maintainable than manually annotated update points used in prior work [1], [5], [9].

To detect per-thread stalling points, the profiler relies on statistical profiling of library calls. Intuitively, a stalling point is simply identified by the blocking call where a given thread spends most of its time during the execution-stalling test workload. To detect per-thread long-lived loops, our profiler relies on standard loop profiling techniques [34], [35]. Intuitively, a long-lived loop is simply identified by the thread's deepest loop that never terminates during the test workload. At the end of the profiling run, our quiescence profiler produces a report with all the (short-lived and long-lived) classes of threads identified, their long-lived loops, and their stalling points.

Each stalling point is automatically classified as *persistent* or *volatile*—that is, whether it is already visible or not right after server initialization—and as *external* or *internal*—that is, whether the corresponding blocking call is listening for external events (e.g., `select`) or not (e.g., `pthread_cond_wait`). In addition, a policy decides how each stalling point participates in our quiescence detection protocol (explained later on in the paper). Three options are possible: (i) *quiescent point*—marks a valid quiescent state for a given thread to actively participate in our protocol; (ii) *blocking point*—allows execution to stall indiscriminately before reaching the next quiescent point; (iii) *cancellation point*—allows returning an error (e.g., `EINTR`) to the program at quiescence detection time. The default policy is to promote all the persistent stalling points to quiescent points and all the volatile ones to blocking points. The rationale is to allow, by default, only the globally quiescent states that can be reconstructed in a fully automated way by *mutable reinitialization* after restart.

Let us now take a closer look at MCR's instrumentation process. MCR's static instrumentation relies on the gathered profile data to instrument all the stalling points identified in the call stack of each long-lived thread in the program. Our instrumentation currently relies on thread-local flags to propagate call stack information to every long-lived blocking call site and instrument stalling points correctly. For this purpose, MCR wraps every blocking library call site corresponding to a profiled stalling point in a way that it allows what we refer to as *unblockification* (i.e., elimination of persistent blocking behavior).

Unblockification exposes the original library call semantics to the program, but guarantees that every wrapped blocking call never truly blocks user-space execution for an extended period of time. Meanwhile, it periodically calls a predetermined hook—which implements MCR's quiescence detection protocol. This design ensures that all the blocking calls are *short-lived* and fully *controllable* by construction at quiescence detection time.

Unblockification meets three key requirements: (i) efficiency; (ii) low CPU utilization; (iii) low quiescence detection latency. Concerning efficiency, unblockification relies on standard timeout-based versions of library calls (e.g., `semtimedop`) and simply loops through repeated call invocations until control must be given back to the program. When a timeout-based version of the call is not available, we resort to either the asynchronous version of the call (e.g., `aio_read`) or—depending on availability—its nonblocking version (e.g., nonblocking `accept`) followed by a generic timeout-based call listening for the relevant events (e.g., `select`). The latter strategy reduces the number of mode switches to the minimum when the program is under heavy load and thus on a performance-sensitive path. Our other requirements highlight the evident tradeoff between unblockification latency and CPU utilization. In other words, short timeouts translate to very fast loop iterations and frequent invocations of our instrumentation hooks, but also to high CPU utilization. To address this problem, our implementation dynamically adjusts the unblockification latency, using low values that guarantee fast convergence at quiescence detection time—currently 1ms—and higher, more conservative values otherwise—currently $100ms$, which resulted in no visible CPU utilization increase in our test programs.

We note that unblockification is a semantics-preserving transformation of the original program which ensures three important properties. First, it guarantees that stalling point execution always revolves around short-lived loops with bounded iteration latency even when a thread is blocked indefinitely. Second, it provides a simple way to enforce our stalling point policies (e.g., allow blocking behavior in case of blocking points or call our hooks at the top of each short-lived loop iteration in case of quiescent points). Third, it can unambiguously identify internal or external events received by long-lived blocking calls (given the well-defined semantics of standard library calls) and pass this knowledge to our hooks at quiescence detection time. These properties all serve as a basis for our quiescence detection protocol.

```
procedure COORDINATOR
    $Q \leftarrow 1$
    repeat
        $A \leftarrow 0$
        SYNCHRONIZE_RCU()
        SYNCHRONIZE_RCU()
    until $A = 0$
    $Q \leftarrow 2$
    SYNCHRONIZE_RCU()
    SYNCHRONIZE_RCU()
```

```
procedure QUIESCENTPOINT
    if $Q > 0$ then
        if $Active$ then
            $A \leftarrow 1$
        if $Initiator$ or $Q > 1$ then
            RCU_THREAD_OFFLINE()
            THREAD_BLOCK()
            RCU_THREAD_ONLINE()
            THREAD_UNBLOCKED()
    RCU_QUIESCENT_STATE()
```

Fig. 2: Quiescence detection protocol pseudocode. The COORDINATOR code runs on a separate thread. The QUIESCENTPOINT is called by application threads when a quiescent point is reached.

4.2 Quiescence detection protocol

Given the instrumentation process described in the previous subsections to identify quiescence points and "unblockify" them, it is possible to detect when the program under analysis reaches a quiescent state and exploit this setting to induce a safe live update.

MCR identifies quiescence through an ad-hoc quiescence detection protocol based on two key observations. First, long-running programs are naturally structured to allow threads waiting for external events (e.g., a new service request or a timeout) to block indefinitely. Second, in the face of no external events, well-formed programs must normally reach a globally quiescent state—all the threads stalling at quiescent points—in bounded time. Building on these observations, our protocol enforces simple *barrier synchronization* for all the threads blocked on external events—that is, *initiator threads*—and waits for all the threads processing internal events—that is, *internal threads*—to complete pending event-processing operations before detecting quiescence. When quiescence is detected, no new event can be initiated by construction and all the threads can be safely blocked at their quiescent points. The key challenge is to automatically determine how long to wait for internal events to complete without blocking threads in a deadlock-prone fashion (or delaying quiescence for an extended period of time).

The naive solution is to scan all the thread call stacks and verify each thread has reached a quiescent point. This strategy, however, is not race free in absence of a consistent view of all the running threads. Worse, even a globally consistent snapshot of all the call stacks is not sufficient in the presence of asynchronous thread interactions. Suppose a thread A signals a thread B blocked on a condition variable and then reaches its next quiescent point. Before B gets a chance to process the event, a global call stack snapshot might mistakenly conclude that both threads are idle at their quiescent points and detect quiescence.

This race condition, known as the *launch-in-transit hazard* [36], is a recurring problem in the *Distributed Termination Detection* (*DTD*) literature [36], [37], [38]. All the solutions to the DTD problem rely on explicit event tracking [37], a costly solution in a local context partially explored in prior work [24]. Fortunately, unlike in DTD, we found that, in a local context, avoiding event tracking is possible, given that local events propagate in bounded time.

In particular, the key idea is to wait for all the threads to reach a quiescent point with no event received since the last quiescent point. This strategy effectively reduces our original global quiescence detection problem to a local quiescence detection problem—that is, quiescing short-lived loop iterations. To address the latter, we rely on RCU [39], a scalable, low-latency, and deadlock-free local quiescence detection scheme. RCU-like solutions to the problem of global quiescence detection were attempted before [22], [28], but in much less ambitious architectures that simply disallowed long-lived threads. Our design is based on the QSBR flavor of `liburcu` [40], the fastest known user-space implementation for local quiescence detection with nearly zero overhead. The implementation provides a `synchronize_rcu` primitive, which allows a *controller thread* to wait for one quiescent period—that is, for all the threads to reach a *quiescent state* at least once from the beginning of the period (i.e., from the moment the `synchronize_rcu` primitive was invoked). A detailed explanation of the QSBR is beyond the scope of the paper and can be found in [40]. In the remainder of this paragraph we focus on the specific aspects of our solution and we complement our explanation with a simple example.

Our RCU-based instrumentation ensures that threads atomically enter a nonquiescent state at creation time (i.e., `pthread_create` blocks waiting for the new thread to complete RCU registration), atomically traverse a quiescent state at each quiescent point right before reaching the designated blocking call, and enter an *extended quiescent state* [40] (i.e., they no longer participate in the quiescence detection protocol) at exit time or when our quiescence detection protocol dictates them. This strategy allows our protocol to transparently deal with an arbitrary number of short-lived and long-lived threads.

Figure 2 illustrates the simplified steps of our quiescence detection protocol in more details, as also described hereafter. The COORDINATOR publishes a quiescence detection protocol event ($Q = 1$) and resets the global active flag (A) to 0. Next, it waits for a first quiescent period to ensure the protocol is visible to both INITIATOR and INTERNAL threads. It then waits a second quiescent period to give *any* instrumented thread a chance to report an active state—whether the last blocking call received an event (or a new thread was created). The entire sequence is repeated until quiescence is detected, that is, no thread changed A in the last quiescent period. In the second phase, the coordinator publishes a barrier event ($Q = 2$) and waits for two more periods, the first period to ensure the barrier event is visible to all the threads and the second period to ensure all the threads react and safely block at their quiescent points.

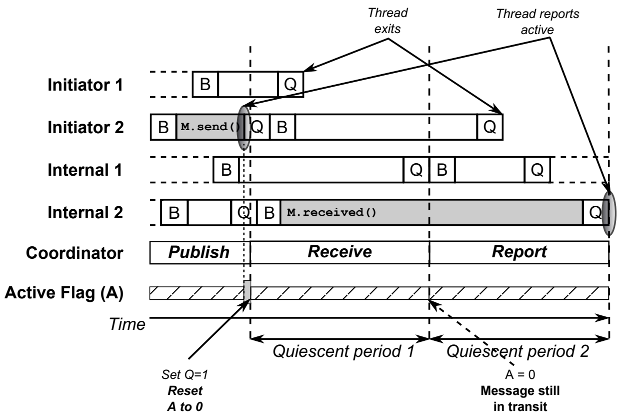


Fig. 3: A sample run of the 1st phase quiescence detection protocol with 2 $Initiator$ threads, 2 $Internal$ threads, and the $Coordinator$ thread. This is a *worst-case* scenario, with a message still in transit after the 1st period, even though no thread has reported being active. B represents *native blocking calls*, and Q represents *quiescent point instrumentation hook* calls.

Our quiescent point instrumentation, in turn, implements the thread-side protocol logic. When the protocol is in progress ($Q > 0$), our hook reports an active state (if any) to the coordinator and blocks the current thread if it is an initiator thread or a barrier event is in progress. For this purpose, lines 6–7 allow the current thread to enter an extended quiescent state and block on a condition variable. Lines 8–9, in contrast, allow the current thread to leave an extended quiescent state and synchronize before resuming execution—in case the coordinator decides to abort the protocol, for example after a predetermined timeout. Note the `rcu_quiescent_state` call at the bottom, the only step executed also during regular execution, to mark all the quiescent state transitions correctly.

By leveraging two common and well-known RCU uses—*publish-subscribe* and *waiting for things to finish* [41]—our protocol provides race-free and deadlock-free quiescence detection in only $2q + 2$ quiescent periods—with $q = 1$ if the program is already globally quiescent and otherwise bounded by the length of the maximum internal event chain.

4.3 A simplified example

Figure 3 exemplifies a run of the first phase of our quiescence detection protocol ($Q = 1$). This sample run depicts a worst-case scenario, with two active threads reacting to the published protocol event only after two quiescent periods due to in-transit messages. In the example, the INITIATOR 2 thread sends a message to the INTERNAL 2 thread, and reports an active state by setting $A = 1$. Immediately after, the COORDINATOR thread publishes a quiescence detection event (*Publish* stage), setting $Q = 1$ and resetting $A = 0$. At the beginning of the first quiescent period (*Receive* stage), both program threads immediately reach a quiescent state—only later followed by the first initiator thread (entering an extended quiescent state) and by the first internal thread (marking the end of the first quiescent period). This ensures that the INTERNAL 2 thread can receive the pending message, but does not prevent the receiving thread from reporting an active state only at a later stage. As a result, at the end of this stage, no threads have reported being active, but in reality one of them is still processing a message. Hence, we need a second quiescent period (*Report* stage) to ensure that the INTERNAL 2 thread can finally report being active ($A = 1$) before reaching a new quiescent state. This allows the COORDINATOR thread to detect nonquiescent behavior and repeat the first phase of our protocol until quiescence is detected. Since INITIATOR threads are structurally prevented from receiving new external events, the protocol converges by construction.

5 MUTABLE REINITIALIZATION

Mutable reinitialization seeks to restore all the updated program threads (and processes) in the new version in the quiescent state equivalent to the one obtained in the old version at update time. This is to complete control migration in the new version and also automatically reinitialize the largest possible portion of the global data structures. To minimize the number of annotations, mutable reinitialization relies on the key observation that running a server program's startup code tends to naturally initialize long-lived server threads (and processes) and converge to a quiescent state that closely matches the one in the old version. When the server model yields stable quiescent states (e.g., in event-driven servers), in particular, this strategy fully automates the entire process with no additional annotations required.

Piggybacking on existing startup code paths, however, raises two challenges: (*i*) how to prevent the startup code from accepting new server requests, which would violate MCR's atomic live update semantics and hamper the ability to rollback failed update attempts; (*ii*) how to allow startup code to complete correctly without disrupting the old version, which is blocked but still active in the background.

Mutable reinitialization addresses the first challenge by allowing a controller thread to reinitiate the quiescence detection protocol before allowing the startup code to run. This strategy forces all the long-lived threads to safely block at their quiescent points without being exposed to new external events. To address the second challenge, mutable reinitialization carefully controls the startup process in the new version by replaying the necessary startup-time operations (i.e., system calls) from the log recorded in the old version, providing the code in the new version with the illusion that the execution of the program is starting up from a fresh state.

Unlike traditional record-replay [42], [43], [44], [45], however, mutable reinitialization does not attempt to deterministically replay execution, a strategy which would otherwise forbid any startup-time changes. The idea is to replay only the operations that refer to *immutable state objects*.

5.1 Handling immutable state objects

Immutable state objects are all the objects that refer to external (e.g., in-kernel) state, which MCR must conservatively inherit and preserve in the new version. In other words, these are the only objects allowed to violate the *mutable* MCR semantics. For example, the file descriptor associated to the server's main listening socket—automatically inherited by MCR at startup—cannot be altered (or recreated) by the startup code or the associated in-kernel state will be lost. Nevertheless, the startup code in the new version may expect such file descriptor to be created and stored in global data structures. Thus, replaying all the operations associated to such file descriptor (e.g., `socket()`) is crucial to allow the new startup code to complete correctly and without disrupting the file descriptor inherited but still *shared* with the old version. The rest of the startup code—potentially very different between versions—in turn, is executed live. Since the replayed operations all refer to state already inherited in the new version by construction, execution can seamlessly transition between live and replay mode without the specialized kernel support required in traditional mutable replay [14], [46]. Our record-replay implementation, in contrast, is simply based on library-level interception of all the startup-time syscalls.

MCR currently supports three main classes of immutable objects based on our experience with real-world server programs: *(i)* file descriptor numbers inherited from the old version—immutable due to the associated in-kernel state; *(ii)* immutable memory object addresses identified by *mutable tracing* (see below)—immutable due to partial knowledge of global pointers; *(iii)* process/thread IDs—immutable since they carry process-specific state potentially stored in global data structures that must be transferred to the new version.

Unfortunately, mapping and preserving immutable objects inherited from the old version at replay time is challenging in a multiprocess context (e.g., a server with one master and one worker process). The problem is exacerbated by the need to avoid unnecessary—and potentially expensive—immutable object tracking during normal execution. Consider the naïve solution for file descriptors of having every process in the new version (e.g., the new worker process) inherit all of its counterparts' (i.e., the old worker process's) file descriptors. There are two major problems with this approach, with similar considerations applicable for other immutable objects as well. First, the multiprocess nature of the startup process may result in an old file descriptor number clashing with another one already inherited from the parent process at `fork` time. Second, file descriptor numbers may be reused during or after startup, which implies that mutable reinitialization can no longer unambiguously determine whether a file descriptor number inherited from the old version matches the one associated to a particular operation in the old startup log. This hampers the ability to establish whether a given operation should be replayed or not in the new version.

Mutable reinitialization addresses both challenges by enforcing two key principles: *global inheritance* and *global separability*. Global inheritance allows the first process in the new version to inherit all the immutable objects from all the processes in the old version before allowing the startup code to run. The idea is to preallocate all the necessary immutable objects to avoid object clashing and progressively propagate all the objects down the process hierarchy in the new version for replay purposes. For example, this translates to a master process inheriting all the old file descriptors at startup and every newly created worker process automatically inheriting all of them as dictated by the `fork` semantics. All the immutable objects that do not participate in replay operations in a given process are simply garbage collected when control migration completes. Global separability, in turn, allows all the immutable objects created at startup to acquire globally unique identifiers, preventing the ambiguity introduced by reuse. For example, this translates to the file descriptor number 10 allocated at startup never being allowed to be reallocated after control migration. Note that preventing reuse is not necessary for immutable objects created after startup, which are not target of replay operations and are simply inherited from the old version with no constraints.

5.2 Matching operations

Mutable reinitialization opts for a conservative matching and conflict resolution strategy when replaying the operations from the startup log recorded in the old version. Syscalls are only automatically replayed when a perfect match is found with the log. For instance, if the startup code in the new version is updated to omit a previously recorded syscall, mutable reinitialization immediately flags a conflict—which results in a rollback if not explicitly resolved by the developer. While more sophisticated record-replay strategies based on best-fit conflict resolution are possible [14], our conservative strategy guarantees correctness of control migration while detecting complex changes that inevitably require developer annotations. Further, since the replay surface is small, we expect unnecessary conflicts caused by startup-time changes to be minimal in practice.

To enforce a conservative matching strategy in presence of reordering of operations due to nondeterminism or arbitrary version changes, mutable reinitialization relies on call stack IDs associated to every operation considered. A call stack ID expresses the context of every recorded (or replayed) system call in a version-agnostic way and is computed by simply hashing all the active function names on the call stack of the thread issuing the system call. Call stack IDs are used to match every system call observed at replay time with the corresponding system call recorded in the old startup log. When a mismatch is found, mutable reinitialization suspends replay operations and immediately flags a conflict. Despite its conservativeness—function renaming between versions may produce different call stack IDs for equivalent operations and thereby introduce unnecessary conflicts—we found this matching strategy to be generally more robust to addition/deletion/reordering of system calls and changes to their arguments than alternative strategies based on global or partial orderings of operations [14]. Finally, mutable reinitialization conservatively flags a conflict when matching system calls are issued with nonmatching arguments between versions. To tolerate benign changes to syscall invocations, however, MCR follows pointers and performs a deep comparison of the arguments similar to [14].

6 Mutable Tracing

Mutable tracing seeks to traverse all the *dirty* global data structures (i.e., state objects) in the old version and remap them to the new version, possibly reallocating and type-transforming updated state objects on the fly. This is to complete state transfer in the new version for all the objects not automatically restored by mutable reinitialization. This strategy raises two challenges: *(i)* how to identify the dirty state objects modified after startup in the old version; *(ii)* how to remap and transfer those objects with minimal manual effort, even with partial knowledge of global pointers.

Mutable tracing relies on *soft-dirty bits* tracking to address the first challenge. This is a lightweight user-level *dirty* memory page tracking mechanism available in recent Linux releases and already adopted by emerging user-space checkpoint-restart techniques for incremental checkpointing purposes [11]. The idea is to first clear all the kernel-maintained soft-dirty bits (associated to each memory page in each process) when program startup completes. This causes the kernel to mark all the memory pages as soft-clean and write-protect them to detect write accesses. As a result, the first memory write issued by the program into a given page after startup causes the kernel to regain control, mark the page as soft-dirty, and unprotect the page again—with no further tracking overhead for subsequent accesses. Finally, at live update time, right after our update-time quiescence detection protocol completes, all the soft-dirty bits are retrieved from the kernel and used to determine all the *dirty* memory pages (and the objects contained) on a per-process basis.

To address the second challenge, in turn, mutable tracing relies on three key observations: *(i)* annotations in prior whole-program state transfer strategies [9], [10] were only necessary to compensate for C's lack of rigorous type semantics. This information is needed to unambiguously identify types in the program state and to implement full-coverage heap traversal, given a set of root pointers. Note that *garbage-collected* languages already have this information at their disposal, which is why we further refer to precise tracing as *GC-style tracing*. Not surprisingly, prior work has already demonstrated that annotationless whole-program state transfer is possible for managed languages like Java [47]; *(ii)* similar problems are well-understood in the garbage collection literature [15], [48], [49]. In particular, the problem of remapping the program state in the face of cross-version type and memory layout changes faces the very same challenges of a *precise* and *moving* tracing garbage collector for C [15]. By precise, we refer to the ability to accurately identify object types, necessary to apply on-the-fly type transformations. By moving, we refer to the ability to relocate objects, necessary to support arbitrary state changes in the new version—induced by type transformations, compiler optimizations, or ASLR (Address Space Layout Randomization). Prior work identified many real-world scenarios in which annotations are necessary in this context, such as: explicit or implicit `unions`, custom allocation schemes, uninstrumented libraries, pointers as integers [15], [32]; *(iii) conservative* garbage collectors are well-known solutions to these problems [16], [17], in that they do not require explicit type information at the cost, however, of being unable to support moving behavior—and thus limiting state transformations in our case.

Mutable tracing combines both *precise* and *conservative* tracing techniques to form a hybrid GC-style heap traversal strategy: it starts from a set of root pointers and precisely traces types and pointers in the face of complete and unambiguous type information, but resorts to a conservative (but less update-friendly) tracing strategy otherwise. For example, when visiting a linked list node allocated by an uninstrumented library, mutable tracing recognizes that no type information is available and conservatively tries to locate and traverse all the possible pointers therein with no assumption on the actual object layout. To implement this strategy for state transfer purposes, MCR gracefully relaxes the original full-coverage data structure transformation requirement. It marks static/dynamic memory objects that are conservatively traversed (and thus cannot be safely relocated after restart) as immutable state objects and raises a conflict when such objects with incomplete or ambiguous type information (e.g., the linked list node in our example) are found changed by the update. This strategy allows the developer to tradeoff the initial annotation effort against the number of update-induced state transformations that can be automatically remapped by mutable tracing without additional annotations. We envision developers deploying an annotationless version of MCR at first, and then incrementally adding annotations only on the data structures that change more often if their experience with the system generates an undesirable number of conflicts. Even when a fully annotated state is desirable from the developer perspective, our conservative strategy can help developers identify missing annotations or other problematic cases.

The next two subsections discuss in details the tracing techniques adopted by MCR.

6.1 Precise tracing

There are two common strategies to implement the precise tracing strategy required by mutable tracing: *(i)* type-aware traversal functions generated by the frontend compiler [9], [48], [49] or *(ii)* in-memory data type tags associated to the individual state objects to define their types [10]. The former is generally more space- and time-efficient, but the latter can better deal with polymorphic behavior and provide more flexible type management. MCR implements the latter strategy to seamlessly switch from precise to conservative tracing as needed at runtime.

MCR's precise tracing strategy operates in each quiescent process in the new version, fully parallelizing the state transfer operations in a multiprocess context. Each process requests a central coordinator to connect to its counterpart in the old version (if any) identified by the same creation-time call stack ID. Once a connection is established with the old process, MCR creates a fast read-only shared memory channel to transfer over all the relocation and data type tags from the old version. Starting from root global and stack objects, MCR traces pointer chains to reconstruct the entire program state in the old version and remap each object found in the traversal to the new version—copying data, reallocating objects, and applying type transformations, similar to [9], [10]. We also allow custom-specified traversal handlers to handle complex semantic state transformations (similar to [10]), as exemplified earlier at the object level in Listing 1.

6.2 Conservative tracing

The conservative tracing strategy adopted by mutable tracing operates obliviously to its precise counterpart. The idea is to first perform a conservative analysis to identify hidden pointers (i.e., pointers not explicitly exposed by the type information available) and derive a number of necessary invariants for the state objects in the old version. Once the invariants are conservatively preserved across versions, state transfer can be simply implemented on top of precise tracing without worrying about hidden pointers and type ambiguity. In particular, our conservative tracing strategy generates two possible invariants for every object in the old version: *immutability*—the object is immutable and cannot be relocated in the new version—and *nonupdatability*—the object cannot be type-transformed by our precise tracing strategy in the new version (a conflict is generated in case of type changes detected).

To identify such invariants, MCR operates similarly to a conservative garbage collector [16], [17], scanning opaque (i.e., type-ambiguous) memory areas looking for *likely pointers*—that is, aligned memory words that point to a valid live object in memory. Objects pointed by likely pointers are marked as immutable and nonupdatable—we could restrict the latter to only interior pointers (i.e., pointers in the middle of an object), but we have not implemented this option yet. Objects that contain likely pointers are marked as nonupdatable—we could restrict the latter to only certain type changes, but we have not implemented this option yet. Note that our strategy is only partly conservative: MCR traverses the program state using our precise strategy by default and switches to conservative mode only when encountering opaque areas. Further, when possible, our pointer analysis uses the data type tag associated to the pointed object to reject illegal (i.e., unaligned) likely pointers.

Our conservative tracing strategy raises two main issues: *accuracy*—how conservative is the analysis in determining updatability coverage—and *timing*—when to perform the analysis. In our experience, the former is rarely a issue in real-world programs. Prior work has reported that even fully conservative GC rarely suffers from type-accuracy problems on 64-bit architectures—although more issues have been reported on 32-bit architectures [50]. Other studies confirm that type accuracy is marginal compared to liveness accuracy [51]. In our context, liveness accuracy problems are only to be expected for uninstrumented allocator abstractions that aggressively use free lists—or other forms of reuse. Nevertheless, these cases can be easily identified and compensated by annotations/instrumentation, if necessary. Also note that, unlike standard conservative GC techniques, accuracy

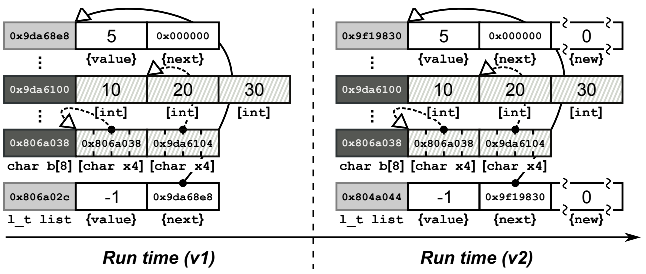


Fig. 4: Example of state mapping using mutable tracing: Array b contains pointers to another array stored as $char$ types which are not captured by *precise* tracing in $v1$, but are *conservatively* preserved in $v2$. While an additional field new is added to the list type, all other fields need to retain their previous values.

problems—that is, likely pointers not reflecting real and live pointers—result only in more immutable objects that MCR cannot automatically type-transform, but not in memory leaks for the running program.

As for the latter, our analysis should be normally performed after the old version has become quiescent. This strategy, however, would normally block the running version for the time to relink the program and prelink the shared libraries to remap nonrelocatable immutable objects (e.g., global variables). Fortunately, we have observed very stable immutable behavior for such objects. As a result, our current strategy is to simply run the analysis and the relinking operations offline. If a mismatch is found after quiescence—although we have never encountered this scenario in practice—we could simply expand the set of immutable objects, resume execution, allow relinking operations in the background, and repeat the entire procedure until convergence is detected.

6.3 A simplified example

Mutable tracing is exemplified in Figure 4, with immutable state objects grayed out and wavy lines highlighting type transformations automatically operated in the new version. In the example, two global objects from Listing 1, the linked list head `list` and the array `b`, are traversed following all the possible pointers to reconstruct the reachable (heap-allocated) data structures. In the case of `list`, mutable tracing relies on the accurate type information available to precisely locate and follow the `next` pointer into the heap-allocated list node in the old version (on the top left). Given the complete knowledge of pointers and types, all the list nodes are marked as mutable and automatically relocated and type-transformed (i.e., with the newly added field `new`) in the new version. The array `b`, in turn, is treated as a generic buffer with unknown type and thus conservatively scanned for possible pointers. In the example, legal pointer values are found to point into a heap-allocated array and `b` itself. Since both values are inherently ambiguous and thus prone to false positives, all the pointed objects in the old version are marked as immutable and forcefully remapped at the same address in the new version.

7 Implementation

We have implemented MCR on Linux (x86), with support for generic server programs written in C. Static instrumentation—implemented in C++ using the LLVM v3.3 API [31]—accounts for 728 (quiescence profiler) and 8,064 LOC [2] (the other MCR components). MCR instrumentation relies on a static library, implemented in C in 4,531 LOC. Dynamic instrumentation—implemented in C in a shared library—accounts for 3,476 (quiescence profiler) and 21,133 LOC (the other MCR components). The `mcr-ctl` tool, which allows users to signal live updates to the MCR backend using UNIX domain sockets, is implemented in C in 493 LOC.

7.1 Profile-guided quiescence

In our implementation, profile-guided quiescence enforces a fast and deadlock-free quiescence detection protocol using `liburcu` [40], a popular user-space RCU library. A current limitation of `liburcu` is its inability to support multiprocess `synchronize_rcu` semantics. To address this issue, MCR uses a process-shared active counter and requests a controller thread in each process to complete the first phase of the protocol. In this phase, newly created processes simply cause the entire protocol to restart. When all the per-process threads complete, MCR transitions to the second phase of the protocol and waits for all the controller threads to report quiescence.

7.2 Mutable reinitialization

In our MCR implementation, as mentioned in Section 5, mutable reinitialization enforces global inheritance and global separability (see Section 5) in different ways for different classes of immutable objects. Immutable static memory objects (e.g., global variables) are inherited using a linker script and naturally guarantee global inheritance and separability by design (no identifier ambiguity possible). Immutable dynamic memory objects (e.g., heap objects) are inherited using *global reallocation* (see below). Separability is enforced by deferring all the `free` operations at the end of startup (no startup-time address reuse) and explicitly flagging startup-time heap objects in allocator metadata (no ambiguity from memory reuse after startup). Immutable file descriptors are inherited using UNIX domain sockets. Separability is enforced by intercepting startup-time file descriptor creation operations (e.g., `open`) to *(i)* allocate new file descriptor numbers in a reserved (non-reusable) range at the end of the file descriptor space and *(ii)* structurally prevent startup-time reuse. Immutable process and thread IDs are handled similarly to file descriptor numbers, except they cannot be simply inherited from the old version. To enforce global inheritance, MCR intercepts startup-time thread and process creation operations (e.g., `fork`) and relies on Linux namespaces [52] to force the kernel to assign a specific ID. This strategy follows the same approach adopted by emerging user-space checkpoint-restart techniques for Linux [11].

A key challenge is how to implement global reallocation of immutable dynamic memory objects, ensuring that each object is reallocated in the new version with the same virtual address as in the old version. MCR addresses this challenge using different strategies, coalescing overlapping memory objects from different processes in the old version into "superobjects" reallocated in the new version at startup (and deallocated later when no longer in use). In particular, shared libraries are copied and prelinked [53] in a separate directory before startup. MCR instructs the dynamic linker to use our copies, allowing the libraries to be remapped at the same virtual address as in the old version. This also allows MCR to reallocate all the dynamically loaded libraries correctly using `dlopen`. Memory mapped objects, in turn, are remapped at the same address using standard interfaces (i.e., `MAP_FIXED`). To provide strong safety guarantees in case of rollback, we also envision memory shared with the old version to be *shadowed* during startup and remapped as expected only at the end, a strategy that our current prototype does not yet fully support. Global reallocation of heap objects poses the greatest challenge, given that standard allocators provide no support for this purpose. MCR addresses this problem by leveraging the intuition that common allocator implementations behave similarly to a buffer allocator for an ordered sequence of allocations in a fresh heap state. MCR implements this strategy for `ptmalloc` [54]—the standard `glibc` allocator—using a single `malloc` arena, but we believe a relatively allocator-independent implementation is possible assuming predictable allocation behavior and `malloc` header size—currently inferred by gaps between dummy allocations performed at startup. We also envision this abstraction to become part of standard allocator interfaces once MCR is deployed—similar to `ptmalloc`'s existing `get_state` and `set_state` primitives for *local reallocation* used in standard checkpoint-restart on a per-process basis.

7.3 Mutable tracing

In our MCR implementation, mutable tracing relies on instrumentation-maintained data type tags to enforce precise tracing behavior and on run-time policies to decide when a traversed memory area must be treated as opaque—thus resorting to conservative tracing.

Similar to prior precise tracing strategies based on

2. Lines of code reported by David Wheeler's SLOCCount.

data type tags [10], [15], MCR relies on static instrumentation to store relocation and data type tags for all the relevant static objects (i.e., global variables, functions, etc.) and change all the allocator invocations to call ad-hoc wrapper functions that maintain relocation and data type tags in in-band allocator metadata. To determine the allocation type on a per-callsite basis, MCR relies on static analysis of allocator operations, similar to [15]. MCR also borrows the tracking technique for generic stack variables, maintaining a linked list of overlay stack metadata nodes [15]. While inspired by prior work, our instrumentation has a number of unique properties. First, ambiguous cases like `unions` require no annotations [10] or tagging [15], given that our tracing strategy can be made conservative when needed. Similarly, MCR does not require full allocator instrumentation for complex allocation schemes. Our allocation type analysis can currently only support standard allocators (i.e., `malloc`) or—using annotations—region-based allocators [55]. For more complex allocator abstractions, our allocation type analysis resorts to fully conservative behavior. Finally, stack variable tracking—expensive at full coverage [15]—is limited to all the functions that quiescence profiling found active on the call stack of some thread blocked at a quiescent point.

To recognize object pairs across versions for remapping purposes (i.e., variable `x` in the old version is to be remapped to variable `x` in the new version), we use a number of strategies dictated by the MCR model. We use symbol names to match static objects and allocation site information to match dynamic objects not automatically reallocated by mutable reinitialization at startup time—which must thus be reallocated at state transfer time. Dynamic objects already reallocated at startup time, in contrast, are matched by their call stack ID, similar to the analogous startup-time operations. Individual program threads, finally, are matched based on their creation-time call stack IDs and all their stack variables remapped using the associated symbol names.

Finally, MCR allows developers to configure run-time policies to identify opaque memory areas that must be traced using a conservative rather than precise tracing strategy, but also provides default values that we found realistic in real-world server programs. The default behavior is to enable conservative tracing for `unions`, pointer-sized integers, `char` arrays, and uninstrumented allocator operations, but different program-driven policies are also possible. Currently, MCR does not conservatively analyze nor transfer shared library state by default, since we have observed that most real-world server programs already reinitialize shared libraries and their state correctly at startup time. Nonetheless, the user can instruct MCR to transfer–and conservatively analyze—the state of particular uninstrumented libraries in an opaque way, when necessary.

8 Violating Assumptions

We report on the key issues that might allow real-world server programs to violate MCR's annotationless semantics—excluding developers extensions required to support complex semantic updates. The intention is to foster future research in the field, but also allow programmers to design more *live update-friendly* (and better) software.

Our profile-guided quiescence strategy might require extra manual effort in the following cases: *(i)* missing stalling points in profile data (i.e., not covered by the test workload)—weakens convergence guarantees; *(ii)* misclassified stalling points in profile data (e.g., an external library call used to synchronize internal events)—weakens convergence or deadlock guarantees; *(iii)* overly conservative stalling point policies (i.e., promoting a semi-persistent stalling point to a blocking point)—weakens convergence guarantees. In our experience, the first two cases are rarely a concern in practice. In contrast, we found the last case to be more common in real-world server programs. We believe, however, that these cases are generally straightforward to detect (i.e., either by running the program or simply inspecting the generated profiling information) and can be easily compensated with simple developers annotations (i.e., a quiescence handler).

Our mutable reinitialization strategy requires extra manual control migration effort when the quiescent state obtained at startup time in the new version does not match the update-time one in the old version. We found this scenario to be relatively common in practice for server programs that dynamically spawn threads and processes on demand. A possible solution is to extend our record-replay strategy to code paths leading to all the possible quiescent points, but this may also introduce nontrivial run-time overhead. While annotations are possible, we believe these cases are generally better dealt with at design time. Purely event-driven servers (e.g., nginx) are an example, with a single possible quiescent state allowed throughout the execution.

Mutable reinitialization might also require extra manual effort in the following cases: *(i)* unsupported immutable objects (e.g., process-specific IDs with no namespace support, such as System V shared memory IDs, stored into global variables); *(ii)* nondeterministic process model (e.g., a server dynamically adjusting worker processes depending on the load); *(iii)* nonreplayed operations actively trying to violate MCR semantics (e.g., a server aborting initialization when detecting another running instance). We believe these cases to be relatively common, the last two in particular—Apache httpd being an example. While the last case is trivial to address at design time, the others require better run-time support and more sophisticated process mapping strategies.

	Quiescence profiling					Updates		Changes			Engineering effort	
	SL	LL	QP	Per	Vol	Num	LOC	Fun	Var	Type	Ann LOC	ST LOC
Apache httpd	2	8	8	5	3	5	10,844	829	28	48	181	302
nginx	1	2	2	2	0	25	9,681	711	51	54	22	335
vsftpd	0	5	5	1	4	5	5,830	305	121	35	82	21
OpenSSH	3	3	3	1	2	5	14,370	894	84	33	49	135
Total	6	18	18	9	9	40	40,725	2,739	284	170	334	793

TABLE 2: Overview of all the programs and updates used in our evaluation. Quiescence Profiling: SL – *Short-lived loops*, LL – *Long-lived loops*, QP – *Quiescence points*, PER – *Persistent* (visible after startup), VOL – *Volatile* (not immediately visible after startup). Engineering effort: Lines of code written to enable *mutable reinitialization* (ANN) and to allow *transferring* of complex memory objects (ST), respectively.

Finally, our mutable tracing strategy shares a number of problematic cases that require extra manual effort with prior garbage collection strategies for C [15]. Examples include storing a pointer on the disk or relying on specialized encoding to store pointer values in memory. In the MCR model, these cases are best described as examples of immutable objects not supported by our run-time system. While seemingly uncommon and generally easy to tackle at design time, we did find 1 real-world program (i.e., nginx) using pointer encoding in our evaluation.

9 Evaluation

We evaluated MCR on a workstation running Linux v3.12 (x86) and equipped with a 4-core 3.0 Ghz AMD Phenom II X4 B95 processor and 8 GB of RAM. For our evaluation, we considered the two most popular open-source web servers—Apache httpd (v.2.2.23) and nginx (v0.8.54). In addition, for comparison purposes, we also considered a popular FTP server—vsftpd (v1.1.0)—and a popular SSH server—the OpenSSH daemon (v3.5p1). Such programs (and versions) have been extensively used for evaluation purposes in prior solutions [1], [7], [9], [24], [29]. We configured our programs (and benchmarks) with their default settings and instructed Apache httpd to use the worker module with two servers and fifty worker threads without dynamically adjusting its process model. We benchmarked our programs using the Apache Benchmark (AB) (web servers), the pyftpdlib FTP benchmark (vsftpd), and the built-in test suite (OpenSSH daemon). We repeated all our experiments 11 times and report the median.

Our evaluation answers four questions: *(i) engineering effort*: how much effort does MCR require? *(ii) performance*: how much overhead does MCR add? *(iii) update time*: what is the MCR update time? *(iv) memory usage*: how much memory does MCR use?

9.1 Engineering effort

To evaluate the engineering effort required to deploy our techniques, we first prepared our test programs for MCR and profiled their quiescent points. To put together an appropriate workload for our quiescence profiler, we used three simple test scripts. The first script—used for the web servers—opens 10 long-lived HTTP connections and issues one HTTP request for a very large file in parallel. The second and third scripts—used for OpenSSH and vsftpd, respectively—open 10 long-lived SSH (or FTP) connections—in authentication/post-authentication state–and, for vsftpd, issue one FTP request for a very large file in parallel. Note that our workload is not meant to be necessarily general—Apache httpd, for instance, supports plugins that can potentially create several new quiescent points—but rather to cover all the quiescent points that we have observed being stressed by the execution of our benchmarks. Quiescence points not discovered by the test workload (e.g., untested Apache httpd plugins) but covered by the real workload may result in deferring quiescence (and updates) in production. In practice, our experience shows that, with some knowledge on the tasks carried out by the server, it is generally straightforward to put together a suitable test workload (typically a much stripped down version of the test suites already included in the original programs).

Next, we considered a number of incremental releases following our original program versions, and prepared them for MCR. In particular, we selected 5 updates for Apache httpd (v2.2.23-v2.3.8), vsftpd (v1.1.0-v2.0.2), and OpenSSH (v3.5-v3.8), and 25 updates for nginx (v0.8.54-v1.0.15)—nginx's tight release cycle generally produces much smaller patches than those of all the other programs considered. We deployed the corresponding live updates incrementally for each program (i.e., updating one release into its next incremental release and simulating periodic live update cycles in production) and tested each update for correctness by (i) comparing the output of our benchmarks before and after the update and (ii) validating the integrity of the updated program state using the time-traveling state transfer technique[3] developed in our prior work [56]. Table 2 presents our findings, with an overview of all the programs and updates considered and the effort required to support MCR.

	Precise pointers						**Likely pointers**					
	Total	**Static**		**Dynamic**		**Lib**	**Total**	**Static**		**Dynamic**		**Lib**
	Ptr	**Src**	**Targ**	**Src**	**Targ**	**Targ**	**Ptr**	**Src**	**Targ**	**Src**	**Targ**	**Targ**
Apache httpd	**2,373**	2,272	2,151	101	219	3	**16,252**	185	2,050	16,067	14,201	1
nginx	**1,242**	1,226	1,214	16	26	2	**4,049**	51	293	3,998	3,755	1
nginx$_{\mathbf{reg}}$	**2,049**	1,226	1,455	823	592	2	**3,522**	51	149	3,471	3,372	1
vsftpd	**149**	148	131	1	4	14	**6**	6	0	0	6	0
OpenSSH	**237**	226	211	11	19	7	**56**	5	16	51	32	8

TABLE 3: Mutable tracing statistics aggregated after the execution of our benchmarks. PTR – total number of pointers found, SRC – total number of pointers residing in *statically* or *dynamically* allocated memory objects, TARG – total number of memory objects allocated *statically*, *dynamically*, or by *shared library* code.

The first six grouped columns summarize the data generated by our quiescence profiler. The first two columns detail the number of short-lived and long-lived thread classes identified during the test workload. The short-lived thread classes detected derive from daemonification (all the programs except vsftpd), initialization tasks (Apache httpd), or executing other helper programs (OpenSSH daemon). The long-lived thread classes detected, in turn, originated a total of 18 quiescent points, divided equally in persistent (*Per*) and volatile (*Vol*)—that is, whether they are already visible or not right after startup. OpenSSH and vsftpd's simple process model resulted in only one persistent quiescent point associated to the master process. All the server programs reported volatile quiescent points with the exception of nginx, given its rigorous event-driven programming model. The quiescent points reported were used as is for our quiescence instrumentation with no extra annotations necessary.

The second group of columns provides an overview of the updates considered for each program and the number of LOC changed by them. As shown in the table, the program changes included in the 40 updates considered account for 40,725 LOC overall. The third group, in turn, shows the number of functions, variables, and types changed (i.e., added, deleted, or modified) by the updates, with a total of 2,739, 284, and 170 changes, respectively. The fourth group shows the engineering effort (LOC) in terms of annotations required to prepare our programs for MCR and the extra state transfer code required by our software updates.

As shown in the table, the annotation effort required by MCR is relatively low. Adding annotations was also greatly simplified by the conflicts flagged by mutable reinitialization and mutable tracing. In all the cases we examined, resolving conflicts was possible with simple annotations. In the general case, the annotation effort to resolve conflicts grows with the complexity of the target program and of the updates. While this may sound intimidating, we believe that the in-depth knowledge of the program acquired by MCR can also be used to suggest the developer common annotations (e.g., a global variable changing its name prefix) to semi-automate the annotation process. We have not, however, attempted this approach given that we found the annotation process relatively streamlined for the programs and updates we considered.

When supporting only persistent quiescent points—corresponding to stable thread configurations automatically reconstructed by mutable reinitialization—in particular, Apache httpd required only 8 LOC to prevent the server from aborting prematurely after actively detecting its own running instance and 10 LOC to ensure deterministic custom allocation behavior. Both changes were necessary to allow mutable reinitialization to complete correctly. Further, nginx required 22 LOC to annotate a number of global pointers using special data encoding—storing metadata in the two least significant bits. The latter is necessary for mutable tracing to interpret pointer values correctly. Extending mutable reinitialization to all the other nonpersistent quiescent points profiled, on the other hand, required an extra 82 LOC for vsftpd, 49 LOC for OpenSSH, and 163 LOC for Apache httpd. In addition, we had to manually write 793 LOC to allow state transfer to complete correctly across all the updates considered. The extra code was necessary to implement complex semantic state transformations that could not be automatically remapped by MCR. Moreover, two of our test programs rely on custom allocation schemes: nginx uses slabs and regions [55], Apache httpd uses nested regions [55]. Extending allocator instrumentation to custom allocation schemes increases updatability, but also introduces extra complexity and overhead on allocator operations. To analyze the tradeoff, we allowed MCR to instrument only nginx's region allocator—precise tracing for slab and nested region allocators is not yet supported in our current MCR prototype—and instructed mutable tracing to produce quiescent-time statistics—for both

3. The system applies the update, and then, before releasing control to the new version, runs a "reverse-update" on the new process, "updating" it with the old binary. The memory regions of the process resulting from this reverse-update operation are then compared naively (i.e., with the equivalent of `memcmp`) to the original process, and if any discrepancies are found, the update is aborted and control is rolled back to the first version.

precisely and conservatively identified pointers—after the execution of our benchmarks (see Table 3).

In the two cases, the table reports the total number of pointers detected (*Ptr*), per-region source pointers (*Src*), and per-region pointed target objects (*Targ*). Objects are classified into *Static* (e.g., global variables, but also strings, which attracted the majority of likely pointers into static objects), *Dynamic* (e.g., heap objects), *Lib* (i.e., static/dynamic shared library objects). We draw three main conclusions from our analysis. First, there are many (23,885) legitimate cases of likely pointers—we sampled a number of cases to check for accuracy—which cannot be ignored at state transfer time. Prior whole-program strategies would delegate the nontrivial effort of handling such cases to the developer. In MCR, such pointers result in a fraction of target objects marked as immutable—0.7%-31.9% for our programs, but heavily program and allocator dependent in general—which MCR can automatically handle with no developer annotations as long as the corresponding data structures are not affected by the update. Second, we note a number of program pointers into shared library state (28+11). This confirms the importance of marking shared library objects as immutable if library state transfer is desired. Finally, our results confirm the impact of allocator instrumentation. Apache httpd's uninstrumented allocations produce the highest number of likely pointers (16,067), with nginx following with 3,998. Our (partial) allocator instrumentation on nginx (nginx$_{reg}$) can mitigate, but not eliminate this problem (3,471 likely pointers). Further, even in the case of a fully instrumented allocator (vsftpd and OpenSSH), we still note a number of likely pointers originating from legitimate type-unsafe idioms (6 and 56, respectively), which suggests annotations in prior solutions can hardly be eliminated even in the optimistic cases.

We now directly compare our results with prior live solutions, when possible. Ginseng [5], Upstare [24], and Kitsune [9] have also updated similar versions of vsftpd. Kitsune [9] can support the largest number of quiescent points (6 manually annotated points) with the lowest state transfer engineering effort (101 LOC over 13 updates). MCR, in turn, can support 5 automatically discovered quiescent points (the difference may stem from the different configurations used) with generally lower state transfer effort (21 LOC over 5 updates). Ginseng requires the lowest one-time annotation effort (50 LOC), but much higher state transfer effort (1092 LOC over 12 updates). MCR, in turn, requires higher one-time annotation effort (82 LOC), but lower than similar whole-program live update solutions such as Kitsune (113 LOC).

Ginseng has also updated similar versions of OpenSSH, with support for only 1 manually annotated quiescent point (compared to MCR's 3 automatically discovered points), comparable one-time annotation effort (50 LOC, compared to MCR's 49 LOC), and higher state transfer effort (784 LOC over 10 updates, compared to MCR's 135 LOC over 5 updates). Kitsune has also analyzed the quiescent behavior of Apache httpd, manually annotating 5 quiescent points. MCR, in contrast, was able to automatically discover 8 quiescent points, providing evidence that our profiling strategy, other than being a fundamental building block for our quiescence detection protocol, can be more effective than manual inspection in identifying quiescent points for complex programs. We cannot directly compare our live update results on Apache httpd and nginx with prior solutions, since deploying updates on such servers has not been attempted before. We can speculate, however, that the extensive use of global pointers, type-unsafe idioms, and application-specific allocators in these servers would be challenging to handle in prior solutions without the help of techniques similar to those supported by MCR.

Overall, we regard MCR as an important step forward over prior solutions [5], [9], [10], [24]: (i) much less annotation effort is required to deploy MCR and support updates; (ii) much less inspection effort is required to identify issues with pointers, allocators, and shared libraries.

	Unblock	**+SInstr**	**+DInstr**	**+QDet**
Apache httpd	0.977	1.040	1.043	1.047
nginx	1.000	1.000	1.000	1.000
nginx$_{\mathbf{reg}}$	1.000	1.175	1.192	1.186
vsftpd	1.024	1.027	1.028	1.028
OpenSSH	0.999	0.999	1.001	1.001

TABLE 4: Run time normalized against the baseline. From left to right, the times reported in each column are cumulative (e.g., the *Quiescence Detection* column also includes the *Unblockification*, *Static Instrumentation* and *Dynamic Instrumentation* times).

9.2 Performance

To evaluate the run-time overhead imposed by MCR, we measured the time to complete the execution of our benchmarks compared to the baseline. We configured the Apache benchmark to issue 100,000 requests and retrieve a 1 KB HTML file. We configured the pyftpdlib benchmark to allow 100 users and retrieve a 1 MB file. In all the experiments, we observed marginal CPU utilization increase (< 3%). Run-time overhead results, in turn, are shown in Table 4. We comment on the results for uninstrumented region allocators first. As expected, unblockification alone (*Unblock*) introduces marginal run-time overhead (2.4% in the worst case for vsftpd). The reported speedups are well within the noise caused by memory layout changes [57]. When unblockification is combined with our static instrumentation (*+SInstr*), the run-time overhead is somewhat more visible (4% worst-case overhead for Apache httpd). The latter

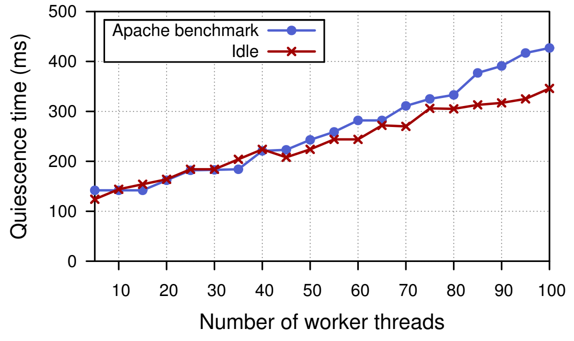


Fig. 5: Quiescence time vs. number of worker threads.

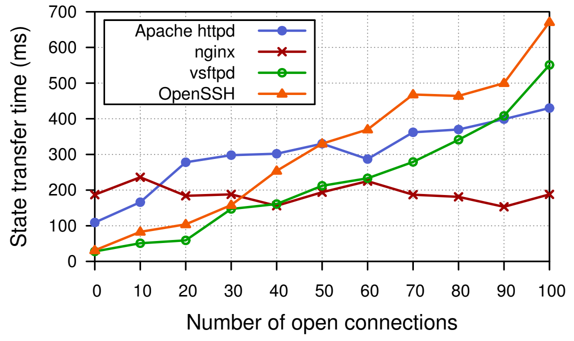


Fig. 6: State transfer time vs. open connections.

originates from our allocator instrumentation, which maintains in-band metadata for mutable tracing. The overhead is fairly stable when adding our dynamic instrumentation (+*DInstr*)—which also tracks all the allocations from shared libraries, other than maintaining process and thread metadata. Finally, our quiescence detection instrumentation (+*QDet*) introduces, as expected, marginal overhead. This translates to the final 4.7% worst-case overhead (Apache httpd) for the entire MCR solution.

To further investigate the overhead on allocator operations, we instrumented all the SPEC CPU2006 benchmarks with our static and dynamic allocator instrumentation. We reported a 5% worst-case overhead across all the benchmarks, with the exception of `perlbench` (36%), a memory-intensive benchmark which essentially provides a microbenchmark for our instrumentation. Our results confirm the performance impact of allocator instrumentation. This is also evidenced by the cost of our region instrumentation on nginx, which incurs 19.2% worst-case overhead (nginx$_{\mathbf{reg}}$ in Table 4). While our implementation may be poorly optimized for nginx's allocation behavior, this extra cost does evidence the tradeoff between the precision of our mutable tracing strategy and run-time performance, which MCR users should take into account when deploying our solution.

Our results show that MCR overhead is generally lower [18] or comparable [1], [5], [9] to prior solutions. The extra costs (unblockification and allocator instrumentation) provide support for automated quiescence detection and simplify state transfer. For example,

	Idle		100 connections	
	Objects	**Dirty**	**Objects**	**Dirty**
Apache httpd	31,494	0.025	36,182	0.151
nginx	5,357	0.076	5,757	0.139
vsftpd	787	0.297	89,487	0.323
OpenSSH	2,525	0.025	269,225	0.198

TABLE 5: Dirty memory objects after our benchmarks.

the tag-free heap traversal strategy proposed in Kitsune [9] would eliminate the overhead on allocator operations, but at the cost of no support for interior or `void*` pointers without extensive annotations.

9.3 Update time

To evaluate the update time—the time the program is unavailable during the update, and thus a measure of the client-perceived latency—we analyzed its 3 main components in detail: *(i)* quiescence time; *(ii)* control migration time; *(iii)* state transfer time. To evaluate quiescence time, we allowed our quiescence detection protocol to complete during the execution of our benchmarks or during idle time. We found that programs with only external quiescent points—vsftpd and OpenSSH—or rarely activated internal points—nginx, whose master process is only activated for recovery purposes—always converge in comparable time in a workload-independent way (around 125 ms, with the first 100 ms attributable to our default unblockification latency), given that our protocol is essentially reduced to barrier synchronization.

Apache httpd is more interesting, with several live internal points interacting across its worker threads. Figure 5 depicts the time Apache httpd requires to quiesce for an increasing number of worker threads, resulting in a maximum quiescent time of 184 ms with 25 threads (default value) and 427 ms with 100 threads (Apache httpd's recommended maximum value). The figure confirms our protocol scales well with the number of threads and converges quickly even under heavy load once external events are blocked. Both properties stem from our RCU-based design.

To evaluate control migration time, we measured the time to complete mutable reinitialization across versions. We found that both the record and replay phase complete in comparable time (less than 50 ms), with modest overhead (1-45%) compared to the original startup time across all our test programs and configurations. Finally, to evaluate state transfer time, we allowed a number of users to connect to our test programs after completing the execution of our benchmarks and measured the time to transfer the state between versions using mutable tracing. Figure 6 depicts the resulting time as a function of the number of open connections at live update time.

Our results acknowledge the impact of the number of open connections on state transfer time, due to

	Static	Run-time	Update-time
Apache httpd	2.187	2.100	7.685
nginx	2.358	4.111	4.656
$\mathbf{nginx_{reg}}$	2.358	4.330	4.829
vsftpd	3.352	5.836	14.170
OpenSSH	2.480	3.047	11.814

TABLE 6: Normalized physical memory usage.

a generally larger heap state and more processes to transfer for programs handling each connection in a separate process—vsftpd and OpenSSH. Compared to recent program-level solutions such as Kitsune [9]—which only evaluated the impact of a single client on the update time—however, Figure 6 shows that MCR scales fairly well with the number of open connections, with an average state transfer time increase of 371 ms at 100 connections, compared to a baseline of between 28-187 ms with no connections. This behavior stems from our parallel state transfer strategy—which operates concurrent state transformations throughout the process hierarchy—and our dirty object tracking strategy—which drastically reduces the amount of state to transfer with essentially no impact on the execution (we observed no steady-state overhead on long-running server programs). Table 5 evaluates the impact of the latter, reporting the total number of memory objects as well as the fraction of dirty objects actually considered for state transfer after executing our benchmarks. As shown in the table, our dirty object tracking strategy is very effective in reducing the number of objects to transfer, with only 2.5%-29.7% of the objects considered in the idle configuration. The effectiveness of our strategy is marginally affected when increasing the number of connections, with 13.9%-32.3% of the objects considered for state transfer with 100 connections.Overall, while generally higher than prior in-place solutions [1], [5]—but comparable and more scalable than prior program-level solutions [9], [24]—we believe our update times to be sustainable for most programs. The benefit is full-coverage (and reversible) multiprocess state transfer able to automatically handle C's ambiguous type semantics.

9.4 Memory usage

MCR instrumentation leads to larger memory footprints. This stems from mutable tracing metadata, process hierarchy metadata, the in-memory startup log, and the required MCR libraries. Table 6 evaluates the MCR impact on our test programs. The static memory overhead (235.2% worst-case overhead for vsftpd) measures the impact of our instrumentation on the original binary size. The run-time overhead (483.6% worst-case overhead for vsftpd), in turn, measures the impact of our instrumentation on the resident set size (RSS) observed at runtime, after startup—we found the overhead to be comparable during the execution of our benchmarks. The update-time overhead, finally, shows the maximum RSS overhead we observed at update time, accounting for an extra running instance of the program and auxiliary data structures allocated for mutable tracing (1,317.0% worst-case overhead for vsftpd).

As expected, MCR requires more memory than prior in-place live update solutions, while being, at the same time, comparable to other whole-program solutions that rely on data type tags such as PROTEOS [10]. A tag-free tracing implementation such as the one adopted in Kitsune [9] would help reduce the overhead in this case as well, but also impose the limitations already discussed earlier. MCR favors annotationless semantics over memory usage, given the increasingly low cost of RAM in these days. Also note that we have not attempted to optimize the occupancy of our tags, which are extremely space-inefficient given that our code is shared across several projects with orthogonal goals. With space optimizations, we believe the nontrivial memory overhead currently incurred by MCR can be significantly reduced and come much closer to that introduced by standard (`malloc`) memory allocators.

10 CONCLUSION

This paper presented *Mutable Checkpoint-Restart* (*MCR*), a new live update solution for generic server programs written in C. MCR's design goals dictate support for arbitrary software updates and minimal annotation effort for real-world multiprocess and multithreaded server programs. To achieve these ambitious goals, the MCR model carefully decomposes the live update problem into three well-defined tasks: *(i)* checkpoint the running version; *(ii)* restart the new version from scratch; *(iii)* restore the checkpointed execution state in the new version. For each of these tasks, MCR introduces novel techniques to significantly reduce the number of annotations and provide solutions to previously deemed difficult problems.

To quiesce all the long-lived program threads at checkpointing time, MCR relies on profile-guided quiescence, a new technique which leverages offline profiling information to implement an efficient, time-bound, and deadlock-free quiescence detection strategy at runtime. To implement control migration at restart time, MCR relies on mutable reinitialization to record-replay startup-time operations and create the illusion that the new version is starting up as similarly to a fresh program initialization as possible. This strategy is also crucial to reinitialize a relevant portion of the program state and thus drastically reduce the state transfer surface, resulting in shorter update times and reduced annotation effort to handle complex state transformations. To implement state transfer for the remaining state objects, finally, MCR relies on mutable tracing to traverse global data structures even with partial type and pointer information, thanks to a carefully balanced combination of precise and conservative GC-style tracing techniques. Our experience with real-world server programs demonstrates that our techniques are practical, efficient, and raise the bar in terms of deployability and maintenance effort over prior solutions.

11 Availability

To foster further research in the field, we have open-sourced our core state transfer framework as part of the mainstream release of the MINIX 3 operating system (http://www.minix3.org), with support for OS-level live update. An open-source implementation of application-level MCR is underway.

Acknowledgments

This work was supported by European Research Council under ERC Advanced Grant 227874.

References

[1] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI*, 2006.
[2] T. Dumitras and P. Narasimhan, "Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system," in *Middleware*, 2009.
[3] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *EuroSys*, 2009.
[4] R. S. Fabry, "How to design a system in which modules can be changed on the fly," in *ICSE*, 1976.
[5] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *PLDI*, 2009.
[6] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *USENIX SEC*, 2005.
[7] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "POLUS: A powerful live updating system," in *ICSE*, 2007.
[8] "Ksplice performance record," http://www.ksplice.com/cve-evaluation.
[9] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," *ACM TOPLAS*, vol. 36, no. 4, Oct. 2014.
[10] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in *ASPLOS*, 2013.
[11] "CRIU," http://criu.org.
[12] C. Hayden, K. Saur, M. Hicks, and J. Foster, "A study of dynamic software update quiescence for multithreaded programs," in *HotSwUp*, 2012.
[13] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Trans. Softw. Eng.*, vol. 22, no. 2, 1996.
[14] N. Viennot, S. Nair, and J. Nieh, "Transparent mutable replay for multicore debugging and patch validation," in *ASPLOS*, 2013.
[15] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, "Precise garbage collection for C," in *ISMM*, 2009.
[16] H.-J. Boehm, "Bounding space usage of conservative garbage collectors," in *POPL*, 2002.
[17] H.-J. Boehm, "Space efficient conservative garbage collection," in *PLDI*, 1993.
[18] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *EuroSys*, 2007.
[19] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *VEE*, 2006.
[20] O. Frieder and M. E. Segal, "On dynamically updating a computer program: From concept to prototype," *J. Syst. Softw.*, vol. 14, no. 2, 1991.
[21] D. Gupta and P. Jalote, "On-line software version change using state transfer between processes," *Softw. Pract. and Exper.*, vol. 23, no. 9, 1993.
[22] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: Challenges and solutions to updating an operating system on the fly," in *USENIX ATC*, 2007.
[23] M. Siniavine and A. Goel, "Seamless kernel updates," in *DSN*, 2013.
[24] K. Makris and R. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *USENIX ATC*, 2009.
[25] S. Ajmani, B. Liskov, L. Shrira, and D. Thomas, "Modular software upgrades for distributed systems," in *ECOOP*, 2006.
[26] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE TSE*, vol. 33, no. 12, 2007.
[27] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE TSE*, vol. 16, no. 11, 1990.
[28] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr, "Providing dynamic update in an operating system," in *USENIX ATC*, 2005.
[29] C. Hayden, E. Smith, E. Hardisty, M. Hicks, and J. Foster, "Evaluating dynamic software update safety using systematic testing," *IEEE TSE*, vol. 38, no. 6, 2012.
[30] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, "Mutable checkpoint-restart: Automating live update for generic server programs," in *Middleware*, 2014.
[31] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[32] C. Giuffrida and A. Tanenbaum, "Safe and automated state transfer for secure and reliable live update," in *HotSwUp*, 2012.
[33] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *CCS*, 2011.
[34] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "Loopprof: Dynamic techniques for loop detection and profiling," in *WBIA*, 2006.
[35] "Poor man's profiler," http://poormansprofiler.org/.
[36] R. F. DeMara, Y. Tseng, and A. Ejnioui, "Tiered algorithm for distributed process quiescence and termination detection," *IEEE TPDS*, vol. 18, no. 11, 2007.
[37] N. Mittal, S. Venkatesan, and S. Peri, "A family of optimal termination detection algorithms," *Distributed Computing*, vol. 20, no. 2, 2007.
[38] P. Johnson and N. Mittal, "A distributed termination detection algorithm for dynamic asynchronous systems," in *ICDCS*, 2009.
[39] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *PDCS*, 1998.
[40] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE TPDS*, vol. 23, no. 2, 2012.
[41] P. E. McKenney and J. Walpole, "What is RCU, fundamentally?" http://lwn.net/Articles/262464.
[42] G. Altekar and I. Stoica, "ODR: Output-deterministic replay for multicore debugging," in *SOSP*, 2009.
[43] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic replay with execution sketching on multiprocessors," in *SOSP*, 2009.
[44] D. Subhraveti and J. Nieh, "Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems," in *SIGMETRICS*, 2011.
[45] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *SIGMETRICS*, 2010.
[46] I. Kravets and D. Tsafrir, "Feasibility of mutable replay for automated regression testing of security updates," in *RESoLVE*, 2012.
[47] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: a VM-centric approach," in *PLDI*, 2009.

[48] F. Henderson, "Accurate garbage collection in an uncooperative environment," in *ISMM*, 2002.
[49] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek, "Accurate garbage collection in uncooperative environments revisited," *Concurr. Comput.: Pract. Exper.*, vol. 21, no. 12, 2009.
[50] M. Hirzel and A. Diwan, "On the type accuracy of garbage collection," in *ISMM*, 2000.
[51] M. Hirzel, A. Diwan, and J. Henkel, "On the usefulness of type and liveness accuracy for garbage collection and leak detection," *ACM TOPLAS*, vol. 24, no. 6, 2002.
[52] E. W. Biederman, "Multiple instances of the global Linux namespaces," in *Linux Symposium*, 2006.
[53] "Prelink," http://people.redhat.com/jakub/prelink.pdf.
[54] "ptmalloc," http://www.malloc.de/en.
[55] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Reconsidering custom memory allocation," in *OOPSLA*, 2002.
[56] A. K. Cristiano Giuffrida, Calin Iorgulescu and A. S. Tanenbaum, "Back to the future: Fault-tolerant live update with time-traveling state transfer," in *USENIX LISA*, 2013.
[57] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009.

Cristiano Giuffrida is an assistant professor in the Computer Systems Group at the Vrije Universiteit Amsterdam. His research interests include operating systems, systems security, and systems reliability. Giuffrida received a PhD from the Vrije Universiteit Amsterdam. He was awarded the Roger Needham Award and the Dennis M. Ritchie Award for the best PhD thesis in Computer Systems (Europe and worldwide) in 2015. Contact him at `giuffrida@cs.vu.nl`.

Călin Iorgulescu is a PhD student in the Operating Systems Laboratory of the Department of Computer Science at École Polytechnique Fédérale de Lausanne. His research interests include operating systems, distributed systems, and systems security. Iorgulescu received an MSc in Parallel and Distributed Computer Systems from the Vrije Universiteit Amsterdam. Contact him at `calin.iorgulescu@epfl.ch`.

Giordano. Tamburrelli is currently an assistant professor at the Vrije Universiteit Amsterdam. Previously he has been Marie Curie Fellow at the USI University in Lugano. He received his PhD from Politecnico di Milano and his M.Sc. degree both from the University of Illinois at Chicago and Politecnico di Milano in a joint degree program. He has active research interests in the areas of modeling and verification of systems and software. Contact him at `g.tamburrelli@cs.vu.nl`.

Andrew S. Tanenbaum is a professor of computer science at the Vrije Universiteit Amsterdam. His research interests focus on operating systems and computer security. Tanenbaum received a BS from MIT and a PhD from the University of California, Berkeley. He is a Fellow of the IEEE and the ACM. He was awarded the USENIX Flame Award in 2008. Contact him at `ast@cs.vu.nl`.