# Mutable Checkpoint-Restart: Automating Live Update for Generic Server Programs

Cristiano Giuffrida
VU University Amsterdam
giuffrida@cs.vu.nl

Călin Iorgulescu
EPFL
calin.iorgulescu@epfl.ch

Andrew S. Tanenbaum
VU University Amsterdam
ast@cs.vu.nl

## 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 live update adoption.

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 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 run-time performance overhead (2% on average) and moderate memory overhead (3.9x on average).

## Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management

## General Terms

Management

## Keywords

Live update, DSU, Record-Replay, Garbage collection



Middleware '14, December 08 - 12 2014, Bordeaux, France
Copyright 2014 ACM 978-1-4503-2785-5/14/12 ...$15.00
http://dx.doi.org/10.1145/2663165.2663328.

## 1. INTRODUCTION

Live update, also known as *dynamic software updating* [42], has growingly gained momentum as a solution to the *update-without-downtime* problem, i.e., deploying software updates without stopping running programs or disrupt their state. Compared to the most common alternative—i.e., *rolling upgrades* [20]—live update systems require no redundant hardware and can automatically preserve program state across versions. Ksplice [10] 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 [21]—existing live update systems still have important limitations. *In-place* live update solutions [7, 10, 19, 41, 42] 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 security patches [4]. Prior *whole-program* live update solutions [23,30], in turn, can efficiently support several classes of updates, but require a nontrivial annotation effort which 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 [2], MCR (i) checkpoints the running version—freezing its execution in a *quiescent state* [28]—(ii) restarts the new version—reinitializing it from scratch in a controlled way—(iii) restores the checkpointed state in the new version—transferring the necessary state (e.g., open connections) from the old version. This is similar, in spirit, to classic checkpoint-restart [2, 3, 5, 9, 27], but the mutability between versions yields a *whole-program* live update strategy with support for arbitrary software updates.

To minimize the annotation effort involved, MCR relies on two key observations. First, while transferring the *entire* execution state between different program versions is a notoriously hard problem [26] and generally requires significant manual effort [23, 30], this is not necessary for real-world server programs which 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 [48] to allow the new version to start up as similarly to a fresh program initialization as possible, while piggybacking on existing code paths to seamlessly reinitialize a relevant portion of the up-

dated 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. Second, when transferring the data structures that are found to have been modified after the old version has completed startup operations—and thus cannot be automatically restored by *mutable reinitialization*—precise knowledge on data types in memory—which generally imposes a nontrivial annotation effort at full coverage [23,30]—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* [44] 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 user-maintained annotations, 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.

To summarize, our contribution is threefold. First, we introduce *mutable reinitialization*, a technique which record-replays startup operations between different program versions and exploits existing code paths to automatically reinitialize the new version, its threads, and a relevant portion of its data structures. Second, we introduce *mutable tracing*, a technique which transfers the remaining data structures between versions using precise (when possible) and conservative (otherwise) GC-style tracing strategies. Third, we demonstrate the effectiveness of our techniques in *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).

## 2. BACKGROUND

In the following, we focus on *local* live update solutions for operating systems and long-running C programs.

***Quiescence detection***. Similar to prior work in the area, MCR relies on *quiescence* [28] 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. Some approaches [38] relax this constraint, but then automatically remapping all the possible program states between versions or simply allowing mixed-version execution [18, 19, 39] becomes quickly intractable without extensive user annotations. Quiescence detection algorithms proposed in prior work operate at the level of individual functions [7, 10, 22, 25] or generic events [12, 13, 23, 41, 42, 45]. The former approach is known for its weak consistency guarantees [23,29] and typically relies on passive *stack inspection* [7, 10, 22, 25] that cannot guarantee convergence in bounded time [38, 39]. The latter approach relies on either update-friendly system design [12,23,45]—rarely an option for existing C programs—or explicit per-thread *update points* [30,38,41,42]—typically annotated at the top of long-running loops. MCR opts for the latter option—using simple barrier synchronization similar to [30]—but, in contrast to prior solutions, relies on a generic *quiescence profiler* to run the target program using a test workload and suggest per-thread (and per-loop) *quiescent points* to the user.

***Control migration***. Similar to prior work in the area, MCR relies on *control migration* [30] as a way to restore all the updated program threads after restart. Prior in-place live update models [7,10,12,18,19,39,41,42], however, provide no support for control migration, implicitly forbidding particular types of updates. Prior whole-program live update models, in turn, implement control migration using system design [23, 45], *stack reconstruction* [38], or annotations [30]. The first option is overly restrictive for many C programs. The second option exposes the user 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 user. 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 quiesce at startup, this strategy can drastically reduce the annotation effort required to complete control migration.

***State transfer***. Similar to prior work in the area, MCR relies on *state transfer* [25] as a way to remap the program state between versions (and applying the necessary data structure transformations) after restart. Prior in-place live update solutions, however, either delegate state transfer entirely to the user [7, 10, 12, 18, 19, 39] or provide simple type transformers with no support for pointer transformations [41, 42]. Such restrictions are inherent to the in-place live update model, which advocates "*patching*" the existing program state in place to adapt it to the new version. Prior whole-program solutions, in turn, either delegate state mapping functions to the user [38, 45] or attempt to reconstruct the state in the new version using *precise* GC-style tracing [23, 30]. 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. OVERVIEW

Figure 1 illustrates the typical MCR workflow. To produce an MCR-enabled version of a server program, users first annotate the program (if necessary, as explained later) and allow our quiescence profiler to run the server under a given test workload. This preliminary step helps the user 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. Building the MCR-enabled version of the program, in turn, requires specifying compiler flags which instruct the GNU gold linker (`ld.gold`) to link the code against our static library (`libmcr.a`) and enable our LLVM link-time

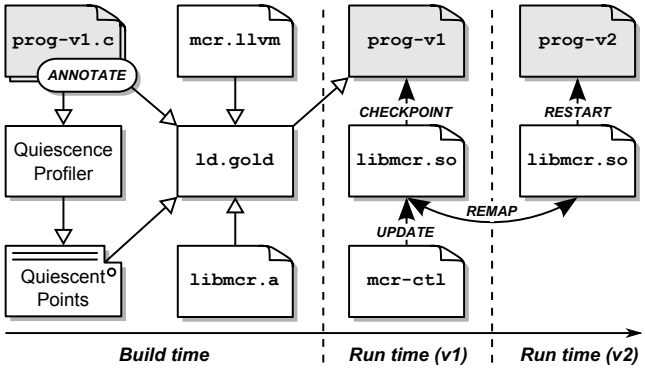


**Figure 1: MCR overview.**

pass [37] (`mcr.llvm`). The latter instruments both the profiled quiescent points for the benefit of our quiescence detection strategy and the program state for the benefit of *mutable tracing*. Running the MCR-enabled version of the program, finally, requires preloading our dynamic instrumentation library (`libmcr.so`), which complements our static LLVM instrumentation with information available only at runtime (i.e., shared libraries) and enables MCR.

During program startup, MCR records all the operations (i.e., system calls) performed by the program in a *startup log*. The latter is later used by *mutable reinitialization* in the new version. After startup, MCR efficiently monitors changes to existing data structures and marks those modified/created after startup as *dirty* in its internal metadata. The latter is later used by *mutable tracing* in the new version.

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 quiescence detection strategy to allow all the program threads to reach a *checkpoint* in a quiescent state. Next, it allows our *mutable reinitialization* strategy to start up the new version from scratch, replay the necessary operations from the old startup log to prevent reexecution errors (e.g., attempt to rebind to port 80), restore the program threads, reinitialize all the startup-time data structures and in-kernel state (e.g., file descriptors). 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 due to arbitrary run-time errors simply causes the new version to terminate and the old version to resume execution from the checkpoint, yielding an atomic and reversible update strategy that hides any live update and rollback event to the clients.

***Server 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. The latter 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`

```
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, user_b_handler);
23 MCR_ADD_REINIT_HANDLER(user_reinit_handler);
```

**Listing 1: A sample MCR-enabled server program.**

(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 [30]. 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 *manipulating* 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 reaches 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, MCR induces the main (and only) program thread to quiesce in `server_get_event` and allows the new program version to independently start up. During startup in the new version, *mutable reinitialization* replays all the necessary operations in `server_init` from the old startup log, inducing code in the new version to naturally reach its first quiescent state in `server_get_event`. With both versions now in an equivalent quiescent state, *mutable tracing* transfers all the *dirty auxiliary* data structures from the old to the new version, omitting the (nondirty) `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 user 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 2). *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 persistent state (e.g., on external storage) [24,30]. The `MCR_ADD_REINIT_HANDLER` annotation, finally, 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, with more complex servers that dynamically spawn threads/processes with long-lived *quiescent points* after startup.

# 4. QUIESCENCE DETECTION

To automatically quiesce a program in a stable and reproducible configuration, MCR requires instrumenting per-thread *quiescent points* that specify safe locations to block long-lived threads—avoiding synchronization and update-safety issues [29]—and yield short call stacks with minimal stack-resident state—avoiding pervasive stack instrumentation to trace stack-allocated data structures, a nontrivial source of overhead in prior solutions [38]. To fulfill both requirements, MCR selects blocking calls (e.g., `accept`) found at the top of long-running thread loops as ideal quiescent point candidates, similar to analogous update point-based strategies adopted in prior work in the area [30, 41, 42].

In contrast to prior work, however, MCR can automatically instrument the target program, profile it using standard profiling techniques, and suggest per-thread quiescent points and the corresponding long-lived loops to the user. To achieve the necessary profiling coverage, our quiescence 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) that must 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.

To detect per-thread quiescent points, our profiler relies on statistical profiling of library calls. Intuitively, a quiescent 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. 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 quiescent points.

MCR's static instrumentation pass relies on all the quiescent points determined to wrap every corresponding blocking library call site in a way that it allows what we refer to as *unblockification*. 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 while periodically calling a predetermined quiescence hook. Our wrappers internally implement this strategy using either asynchronous (e.g., `aio_read`) or timeout-based (e.g., `semtimedop`) versions of standard blocking library calls. When quiescence is requested, MCR's current quiescence hook implementation enforces a barrier synchronization protocol to immediately block all the running program threads, a simple strategy that prior studies on server programs have shown effective in the practical cases of interest [28].

# 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-time 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 the startup code to complete correctly without clashing or 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 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 program is starting up from a fresh state.

Unlike traditional record-replay [8, 36, 43, 46], 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* (e.g., file descriptors), that is objects which MCR sets out to inherit from the old version at startup and cannot be mutated between versions to guarantee correctness and reversibility (i.e., rollback) of the live update process (see below). 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 [35, 48]. Our record-replay implementation, in contrast, is simply based on library-level interception of all the startup-time syscalls.

***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 user. While more sophisticated record-replay strategies based on best-fit conflict resolution are possible [48], our conservative strategy guarantees correctness of control migration while detecting complex changes that inevitably require user 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 [48]. 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 [48].

***Immutable state objects***. In the new version, *Mutable reinitialization* replays only the startup-time syscalls operating on *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.

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 on 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 naive solution for file descriptors—but similar considerations apply to other immutable objects as well—which would allow every newly created process (e.g., the new worker process) in the new version to simply inherit all the file descriptors from its old counterpart (i.e., the old worker process). There are two major problems with this approach. 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 allowed to be reallocated after control migration completes.

MCR enforces these properties 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 (nonreusable) 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 [15] 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 [2].

***Global reallocation***. 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 [34] 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. Heap objects, finally, require dedicated allocator support to enforce a given memory layout in a fresh heap state. MCR implements this strategy for `ptmalloc` (`glibc` allocator).

# 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 on global pointers.

To address the first challenge, *mutable tracing* relies on *soft-dirty bits* tracking, 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 [2]. 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, *mutable tracing* relies on three key observations. First, annotations in prior whole-program state transfer strategies [23,30] 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 GC-style heap traversal, given a set of root pointers. Not surprisingly, prior work has already demonstrated that annotationless whole-program state transfer is possible for managed languages like Java [47]. Second, similar problems are well-understood in the garbage collection literature [11, 31, 44]. In particular, the problem of remapping the program state in face of cross-version type and memory layout changes faces the very same challenges of a *precise* and *moving* tracing garbage collector for C [44]. 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. 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 [24, 44]. Third, *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—

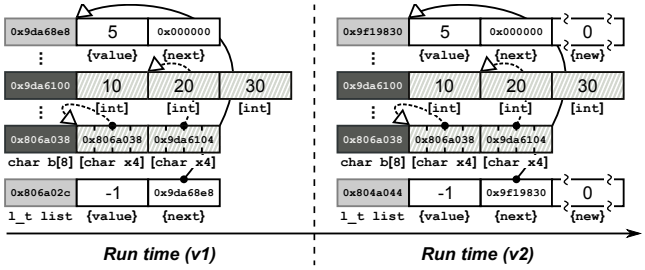


**Figure 2: State mapping using mutable tracing.**

and thus limiting state transformations in our case.

Building on these observations, *mutable tracing* relies on a hybrid GC-style heap traversal strategy, which starts from a set of root pointers and *precisely* traces types and pointers in 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, marking static/dynamic memory objects that are conservatively traversed (and thus cannot be safely relocated after restart) as *immutable state objects* and raising 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 user 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 users deploying an annotationless version of MCR at first, and then incrementally add 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 user perspective, our conservative strategy can help the user identify missing annotations or other problematic cases.

Our *mutable tracing* strategy is exemplified in Figure 2, 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, i.e., 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 on 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.

***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 [11,30,31] or (ii) in-memory data type tags associated to the individual state objects to define their types [23]. 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.

Similar to prior precise tracing strategies based on data type tags [23, 44], 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 [44]. MCR also borrows the tracking technique for generic stack variables, maintaining a linked list of overlay stack metadata nodes [44]. While inspired by prior work, our instrumentation has a number of unique properties. First, ambiguous cases like `unions` require no annotations [23] or tagging [44], 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 [14]. For more complex allocator abstractions, our allocation type analysis resorts to fully conservative behavior. Finally, stack variable tracking—expensive at full coverage [44]—is limited to all the functions that quiescence profiling found active on the call stack of some thread blocked at a quiescent point.

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 as needed, similar to [23, 30]. We also allow user-specified traversal handlers to handle complex semantic state transformations (similar to [23]), as exemplified earlier at the object level in Listing 1.

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.

***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.

Run-time policies decide when a traversed memory area must be treated as opaque. Our default is to do so 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.

Our conservative tracing strategy raises two main issues: *accuracy*—i.e., how conservative is the analysis in determining updatability coverage—and *timing*—i.e., 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 [32]. Other studies confirm that type accuracy is marginal compared to liveness accuracy [33]. 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 problems—i.e., likely pointers not reflecting real and live pointers—result only in a larger number of immutable objects that MCR cannot automatically type-transform, but not in the introduction of memory leaks.

As for the latter, our analysis should be normally performed after quiescing the old version. 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.

# 7. VIOLATING ASSUMPTIONS

We report on the key issues that might allow server programs found "*in the wild*" to violate MCR's annotationless semantics—excluding user 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 quiescence detection strategy might require extra manual effort when the test workload used for profiling purposes fails to cover some quiescent points. While this is subject to the quality of the workload, our experience with real-world server programs shows that the number of quiescent points is generally limited and relatively simple to cover in practice.

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.

Finally, our mutable tracing strategy shares a number of problematic cases that require extra manual effort with prior GC strategies for C [44]. 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 easy to tackle at design time, we did find 1 real-world program (i.e., nginx) using pointer encoding in our evaluation.

# 8. EVALUATION

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 [37]—accounts for 728 (quiescence profiler) and 8,064 LOC [1] (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.

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)—and, for comparison purposes, a popular FTP server—vsftpd (v1.1.0)—and a popular SSH server—the OpenSSH daemon (v3.5p1). We configured our programs (and benchmarks) with their default settings and instructed Apache httpd to use the worker module with 2 servers and 50 worker threads without dynamically adjusting its process model. We benchmarked our programs using the Apache benchmark (AB) [1] (web servers), the pyftpdlib FTP benchmark [6] (vsftpd), and the built-in test suite (OpenSSH daemon). We repeated all our experiments 11 times and report the median.

Our evaluation answers 4 key questions: (i) *Engineering effort*: How much effort does MCR require? (ii) *Performance*: Does MCR yield low overhead? (iii) *Update time*: Does MCR yield reasonable update time? (iv) *Memory usage*: How much memory does MCR use?

***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 a number of 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 a number of 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. 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.
Next, we considered a number of 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. Table 1 presents our findings, with an overview of all the programs and updates considered and the effort required to support MCR.

The first six grouped columns summarize the data gener-

[1]Lines of code reported by David Wheeler's SLOCCount.

| | 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 1: Overview of all the programs and updates used in our evaluation.**

| | 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$_{reg}$** | **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 2: Mutable tracing statistics aggregated after the execution of our benchmarks.**

ated 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 `exec()`ing 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 1 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 two grouped columns provide 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, finally, 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 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. 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 2 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 [14], Apache httpd uses nested regions [14]. 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—slabs and nested regions are not yet supported by our current MCR prototype—and instructed mutable tracing to produce quiescent-time statistics—for both precisely and conservatively identified pointers—after the execution of our benchmarks (Table 2).

In the two cases, the table reports the total number of pointers detected (*Ptr*), also classifying them by source (*Src*) and target (*Targ*) memory region. The source and target memory regions are further classified into *Static* (e.g., global variables, but also strings, which attracted the majority of likely pointers into static objects), *Dynamic* (e.g., heap), *Lib* (i.e., static/dynamic shared libraries). 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 user. In MCR, such pointers result in a fraction of target objects marked as immutable—0.7%-31.9% for our programs, but heavily program/allocator dependent in general—which MCR can automatically handle with no user 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 im-

| | 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 3: Run time normalized against the baseline.**

portance 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. Overall, we regard MCR as an important step forward over prior solutions [23, 30, 38, 41]: (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.

***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 (i.e., $< 3\%$). Run-time overhead results, in turn, are shown in Table 3. We comment on 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 [40]. 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 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 3). While our implementation may be poorly optimized for nginx's allocation behavior, this extra cost does

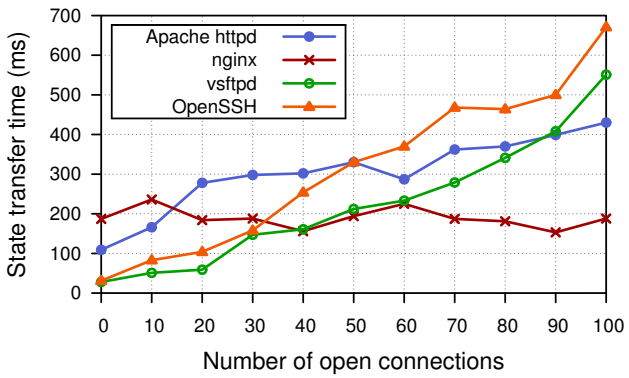


**Figure 3: State transfer time vs. open connections.**

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 [39] or comparable [30, 41, 42] to prior solutions. The extra costs (unblockification and allocator instrumentation) provide support for automated quiescence detection and simplify state transfer. For example, the tag-free heap traversal strategy proposed in Kitsune [30] would eliminate the overhead on allocator operations, but at the cost of no support for interior or `void*` pointers without pervasive user annotations.

***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 three 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. We found that all our programs always converge in comparable time (less than 100 ms) and in a workload-independent way, confirming results reported in prior work for similar barrier synchronization-based quiescence protocols [30].

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 3 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 a generally larger heap state and more processes to transfer for programs handling each connection in a separate process—i.e., vsftpd and OpenSSH. Compared to recent program-level solutions such as Kitsune [30]—which only evaluated the impact of a single connection on the update time—however, Figure 3 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 through-

out the process hierarchy—and our *dirty* object tracking strategy—which drastically reduces the amount of state to transfer (68%-86% reduction with 100 connections).

Overall, while generally higher than prior in-place solutions [41,42]—but comparable and more scalable than prior program-level solutions [30,38]—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.

***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. In detail, we measured a binary size overhead of 118.7%-235.2% and a run-time resident set size (RSS) overhead of 110.0%-483.6% (288.5% on average) when running our benchmarks.

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 [23]. A tag-free tracing implementation such as the one adopted in Kitsune [30] 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.

# 9. 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 user annotations and provide effective solutions to previously deemed difficult problems. To quiesce all the long-lived program threads at checkpointing time, MCR relies on standard profiling techniques to identify the per-thread quiescent points in the program and implement a simple barrier synchronization protocol. 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 server programs found "*in the wild*" demonstrates that our techniques are practical, efficient, and raise the bar in terms of deployability and maintenance effort over prior solutions.

# 10. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments. This work was supported by European Research Council under grant ERC Advanced Grant 2008 - R3S3.

# 11. REFERENCES

[1] Apache benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] CRIU. `http://criu.org`.

[3] Cryopid2. `http://sourceforge.net/projects/cryopid2`.

[4] Ksplice performance on security patches. `http://www.ksplice.com/cve-evaluation`.

[5] OpenVZ. `http://wiki.openvz.org`.

[6] pyftpdlib. `https://code.google.com/p/pyftpdlib`.

[7] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.*, pages 19–19, 2005.

[8] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 193–206, 2009.

[9] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. of the IEEE Int'l Symp. on Parallel and Distributed Processing*, pages 1–12, 2009.

[10] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth ACM European Conf. on Computer Systems*, pages 187–198, 2009.

[11] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurr. Comput.: Pract. Exper.*, 21(12):1572–1606, 2009.

[12] 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 *Proc. of the USENIX Annual Tech. Conf.*, pages 1–14, 2007.

[13] 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 *Proc. of the USENIX Annual Tech. Conf.*, page 32, 2005.

[14] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proc. of the 17th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, 2002.

[15] E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proc. of the Linux Symposium*, 2006.

[16] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 197–206, 1993.

[17] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–100, 2002.

[18] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew.

Live updating operating systems using virtualization. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments*, pages 35–44, 2006.

[19] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Eng.*, pages 271–281, 2007.

[20] T. Dumitras and P. Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proc. of the 10th Int'l Conf. on Middleware*, pages 1–20, 2009.

[21] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proc. of the Second Int'l Conf. on Software Eng.*, pages 470–476, 1976.

[22] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *J. Syst. Softw.*, 14(2):111–128, 1991.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 279–292, 2013.

[24] C. Giuffrida and A. Tanenbaum. Safe and automated state transfer for secure and reliable live update. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades*, pages 16–20, 2012.

[25] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Softw. Pract. and Exper.*, 23(9):949–964, 1993.

[26] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[27] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[28] C. Hayden, K. Saur, M. Hicks, and J. Foster. A study of dynamic software update quiescence for multithreaded programs. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades*, pages 6–10, 2012.

[29] C. Hayden, E. Smith, E. Hardisty, M. Hicks, and J. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Softw. Eng.*, 38(6):1340–1354, 2012.

[30] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations*, 2012.

[31] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. of the 3rd Int'l Symp. on Memory management*, pages 150–156, 2002.

[32] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *Proc. of the 2nd Int'l Symp. on Memory Management*, pages 1–11, 2000.

[33] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, 2002.

[34] J. Jelinek. Prelink `http://people.redhat.com/jakub/prelink.pdf`.

[35] I. Kravets and D. Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[36] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 155–166, 2010.

[37] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, page 75, 2004.

[38] K. Makris and R. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.*, pages 397–410, 2009.

[39] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second ACM European Conf. on Computer Systems*, pages 327–340, 2007.

[40] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2009.

[41] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–24, 2009.

[42] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 72–83, 2006.

[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 *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 177–192, 2009.

[44] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for C. In *Proc. of the Int'l Symp. on Memory management*, pages 39–48, 2009.

[45] M. Siniavine and A. Goel. Seamless kernel updates. In *Proc. of the 43rd Int'l Conf. on Dependable Systems and Networks*, 2013.

[46] D. Subhraveti and J. Nieh. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 109–120, 2011.

[47] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–12, 2009.

[48] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 127–138, 2013.