MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution

Paper # 206

Abstract

Dynamic Software Updating (DSU) is a technique by which to update running software without shutting it down, with the goal of ensuring timely updates and non-stop service. However, applying the update could induce a long delay in service, and bugs in the update—both in the changed code and in the specification for effecting that change dynamically—may cause the updated software to crash or misbehave.

This paper proposes MVEDSUA,[1] a system that solves these problems by augmenting a DSU system with support for Multi-Version Execution (MVE). To start, MVEDSUA performs the update in parallel with the original system, thereby avoiding any service delay. Then, it confirms that the updated and original systems agree when given the same inputs. Expected differences can be easily specified by the programmer, so remaining differences signal likely errors. If the new version shows no problems, it can be installed permanently.

We implemented MVEDSUA on top of Kitsune, a modern DSU system, and Varan, a modern MVE system, and used it to update several high-performance servers, Redis, Memcached, and Vsftpd. Our results show that MVEDSUA significantly reduces the update-time delay, imposes little overhead in steady state, and easily recovers from a variety of update errors.

1 Introduction

For many modern software systems, constant availability is a requirement. At the same time, such systems are often subject to frequent updates, including security patches and feature improvements. Applying such updates by stopping, patching, and restarting the patched system compromises its availability. Rolling upgrades can mask downtime in a distributed system [7] by gracefully directing new connections away from to-be-upgraded nodes, so they can be patched and restarted with less disruption. But this approach still drops the non-persistent state of each node (long-running connections, and in-memory state), degrading service. Many applications—including (in-memory) databases, FTP and SSH servers, gaming servers, and social media platforms—would prefer to keep in-memory state across upgrades. For example, this problem is acute enough that Facebook uses a custom Memcached that keeps in-memory state in a ramdisk to which it reconnects on restart, after an update [23].

Dynamic software updating (DSU) is an approach that enables applications to benefit from a constant stream of updates without requiring stop-and-restart to apply them. DSU typically works by updating a process *in place*, patching the existing code and transforming the existing in-memory state to an equivalent representation that is compatible with the new code. The research community has developed general-purpose DSU systems [6, 8, 10–12, 15, 22, 29, 35, 37] that support release-level changes to substantial applications, including operating systems, data management systems, and various servers.

While the intention is for a dynamic update to improve the system while keeping it running, there is a chance things could go wrong. It may be that the change itself is buggy—many patches that aim to fix bugs end up introducing new ones [38]. It may also be that the change is correct, but the mechanics of applying it at run-time are the problem [14, 28]. Program code assumes the execution state adheres to a certain format, so changing the code at run-time requires the programmer (often with automated assistance) to define corresponding *transformations* to the execution state, both *control* (i.e., thread call stacks and program counters), and *data* (i.e., contents and format of heap objects). These transformations could have bugs that cause the program to fail. Even if transformations are correct, they might take a long time to perform if the state to transform is very large (e.g., heap), temporarily halting service.

As a solution to these problems, this paper presents MVEDSUA, a novel marriage between a DSU system and a system for *multi-version execution* (MVE). An MVE system works by running multiple instances of a program at once, ensuring that each sees the same set of inputs and confirming that they produce the same (or equivalent) outputs. One modern MVE system is Varan [17], which operates using a lightweight, in-memory record and replay scheme. In this scheme, one version runs as

[1]MVEDSUA is a combination of “MVE” and “DSU” and is pronounced “Medusa”.

a *leader*, recording the results of system calls in an in-memory ring buffer, while the other versions run as *followers*, reading the result on system calls from the ring buffer. Varan works directly on binaries and imposes a small runtime overhead in many practical situations.

MVEDSUA combines Kitsune [15], a DSU system for C programs, with Varan. MVEDSUA execution follows three stages. Initially, MVEDSUA uses Varan to run a Kitsune-enabled version of the software as a single leader: *single leader stage*. When an update becomes available, MVEDSUA creates a new follower by forking the single leader and initiating the update on the follower: *outdated leader stage*. Doing this provides the immediate benefit that there is no pause in service while the update takes place. Once the update is complete, the leader continues to run the old version, while the follower runs the new version. If their behavior disagrees, this could signal a bug in the new version or the update, so we terminate the follower. Of course, some behavioral differences are to be expected. As such, the programmer is able to specify expected differences using Varan's DSL for mapping equivalent system call sequences between versions [27]. Once the divergence is fixed (whether by fixing the code, update, or syscall mapping), we can retry the update, repeating the process until the program is running stably. At this point, we may wish to promote the new version to be the leader, making the old version the follower: *updated leader stage*. Still, eventually, we will discard the follower and run just the new version, reverting back to the single leader stage.

Consider the benefits of MVEDSUA compared to alternative solutions. First, MVEDSUA provides a simple approach to minimizing the update-time pause by performing the update in parallel with service provision. DSU systems that support on-demand state transformation provide a similar benefit, but are far more complex [22, 26, 29]. Second, MVEDSUA effectively responds to update failures. The simplest alternative to MVEDSUA is simple restart, post failure, but this (a) kills existing connections, (b) discards all accumulated in-memory state, and (c) delays new connections until the program is restarted. One could mitigate (b) somewhat by using *checkpoint/rollback* support [18, 24, 30, 32] to checkpoint the program just before attempting a DSU. If that update fails, we roll back to the pre-update state [12]. However, if the failure happens once execution resumes, rolling back drops all modifications to the state that happened post-update, pre-failure. Another possible mitigation is to roll back by dynamically updating the new version of the program back to the old one, using a programmer-provided, backward transformation [8]. But this is unsatisfying because it piggybacks on the very task (data/control transformation) that could have originated the bug. It also may not always be possible, depending on the change. Section 8 discusses related work in depth.

We have used MVEDSUA to perform multiple dynamic updates on three high-performance servers: Memcached (2 updates), Redis (3), and Vsftpd (13). The added programmer effort of using MVEDSUA was modest: No DSL rules were needed for either Memcached update, one was needed for Redis, and around one was needed for each Vsftpd update, on average. Only Memcached required interesting code changes (about 100 lines) to work properly with MVEDSUA, owing to its use of multiple threads and stateful libraries (see §6.3). In terms of run-time overhead, we found that MVEDSUA imposes 3-9% overhead on throughput during normal operation, as compared to 0-3% for Kitsune on its own. While MVEDSUA is monitoring both the original and updated versions, the overhead is 25-52% (which essentially matches Varan's overhead). Finally, we observed that MVEDSUA can completely eliminate the pause due to updating, and that it can discover and recover from a variety of errors, including those due to failed update specifications, and failures in the updated code itself.

In summary, the main contributions of this paper are:

(1) A novel approach that combines DSU and MVE to hide the update latency of dynamic updates, and tolerate a variety of errors in the update process, including those from new bugs introduced by the new software version, and errors specific to the DSU process. MVEDSUA can tolerate both generic (e.g., crashes) and semantic bugs, by using the old version as an oracle for the common functionality between the two versions.

(2) A prototype implementation of MVEDSUA, using Kitsune [15] and Varan [17], together with an empirical evaluation on the high-performance servers, Redis, Memcached and Vsftpd. Our evaluation on more than a dozen updates shows that MVEDSUA can effectively reduce update latency while incurring only a modest overhead of 3% to 9% in steady state. For these systems, MVEDSUA can also correctly abort the update when it encounters errors caused by bugs introduced in the new version and in the specification of DSU, and can tolerate expected divergences in behavior across versions.

2 DSU and Availability

Dynamic software updating (DSU) is a technology that aims to maintain high application availability in the face of regular and wide-ranging application updates. It typ-

```
1  typedef char* str;
2  struct entry {str key; void* val;};
3  struct entry **table;
4  pthread_mutex table_mutex;
5
6  struct entry *get(str key) { ... }
7  void put(str key, void *val) { ... }
8
9  void clientloop(int cl_fd) {
10   while (1) {
11     kitsune_update(”client”);
12     /* ...process requests on cl_fd */
13   }
14 }
15 main() {
16   kitsune_automigrate();
17   if (!kitsune_is_updating()) {
18     table = malloc(sizeof(struct entry*)*SIZE);
19     for (int i=0; i<SIZE; i++) {
20       table[i] = malloc(sizeof(struct entry));
21       table[i]−>key = ””;
22     }
23   }
24   int l_fd = listen(...);
25   while (1) {
26     kitsune_update(”main”);
27     int cl_fd = get_conn(l_fd);
28     pthread_create(..., clientloop, cl_fd);
29   }
30 }
```

Figure 1: A multithreaded, in-memory key-value store made updateable by Kitsune.

ically works by updating a running application's code in place and transforming its current execution state to be compatible with that new code. This section begins by explaining how DSU works, focusing on the Kitsune DSU system [15] as an example. The section concludes by describing how DSU's promise of high availability is compromised by errors in dynamic updates, as well as long pauses induced by state transformations.

2.1 Dynamic Software Updating

DSU systems usually perform updates in two steps. First, they dynamically load new code, which constitutes the logical modifications, additions, and deletions to the code of the running program. Second, they transform the program's execution state into a form that is compatible with the updated code. For example, an abstraction may change representations between versions (e.g., a set previously represented as a tree is now represented as a hash table), and a transformation is needed to modify in-memory state to conform to the new representation. Such a transformation is application-specific and so often requires some assistance by the programmer. Control state may also change between versions. For example, new variables might be allocated on the stack, and new control paths might be enabled. Mapping the running program's control state to one compatible with new code also often requires programmer assistance.

```
typedef void* typ;
typ string = (void*)0, number = (void*)1; date = (void*)2;
struct entry {str key; void* val; typ t;};

void put(str key,void *val, typ t) { ... }
typ type(str key) { ... }
```

Figure 2: Update to the key-value store in Figure 1.

To be concrete, let us consider how the Kitsune DSU framework [15] could be applied to the C code in Figure 1, which implements the core of a key-value store. (For now, ignore the calls to kitsune_ functions.) The store is contained in the global structure table, which is dynamically allocated to contain SIZE entries at the start of main. Each entry is individually allocated and stored in the table, with a key field set to the empty string. The main loop repeatedly accepts TCP connections from clients, spawning a clientloop thread to process requests to get and update elements of the store. Such client requests are in the form of text commands, such as `PUT balance 1000` and `GET balance`, which (respectively) are translated by clientloop to calls to put and get.

Figure 2 shows an update to this program: each store entry is extended with a t field that indicates the value's type; some standard types string, number, and date are defined. This change impacts the signature of put, which now takes the typ as an argument. The clientloop must change to expose the new API to clients. Client requests like `PUT balance 1000` are translated by clientloop to calls put(”balance”,”1000”,string), i.e., string is the default typ argument. A request `PUT-number balance 1000` specifies the type explicitly; this one translates to a call put(”balance”,”1000”,number). The request `GET balance` would translate to get(”balance”) as before, while the request `TYPE balance` would translate to type (”balance”), returning the key's type.

A Kitsune-enabled program is compiled to a single shared object file, which is loaded by a small driver program. The driver calls into the program's main and begins execution. At start-time the program is in *normal mode* so a call to kitsune_automigrate is a no-op, and a call to kitsune_is_updating returns 0. For our example, the latter call on line 17 means that table and its entries are properly allocated on lines 18–22.

After some time, when an update becomes available, the driver is signaled to load the updated program as a new shared object. To switch execution to that new code, Kitsune begins the process of *quiescing* the running threads. This works by pausing the threads as they reach *update points* manually placed in the code by the programmer; in our example, these are at lines 11 and 26. Once all threads are blocked at an update point, they long-jump back to the start of their entry points (main or a thread's initial function) and then invoke the corresponding entry functions in the newly loaded code in *update mode*. In this mode, a kitsune_automigrate call is no longer a no-op but instead triggers various state transformations specified in the update. A kitsune_is_updating call now returns 1.

For our example, executing the new main (which is the same as the one in Figure 1) calls kitsune_automigrate which causes the contents of table to be transformed: existing two-field entries are freed and replaced by three-field entries. How these are initialized is determined by the programmer. For example, the two fields from the old version might be carried over, and the new field might be initialized with a default value, say string. Next, the conditional on line 17 will be false, so lines 18–22 are *not* executed, thus ensuring the existing (and now transformed) table contents are not overwritten. When a thread reaches the same update point at which it was quiesced originally (as determined by the string argument to its kitsune_update call) it "checks in" and blocks. Once all threads have checked in, they resume execution in normal mode—the update is complete.

The programmer plays a key role in supporting DSU, with Kitsune and other systems. The programmer specifies state transformations and when they take place. Kitsune provides a domain-specific language to assist in specifying state transformations, which are seeded based on a diff of the old and new code; typically a programmer will only have to write a few dozen lines of code for a typical release-level update. The programmer also controls the timing of an update through the use of update points. These should be reached frequently during program execution, and should identify locations at which operations are not in flight (e.g., not holding locks) [15, 29]. For instance, as in our example, the start of event handling loops is a good candidate for such an update point. Some systems attempt to find update points automatically [20, 22, 35]. Unfortunately, the problem is undecidable [13], which leads to spurious update failures simply due to performing an update at the wrong time [14, 28]. Kitsune's mechanism of restarting updated threads and running them in *update mode* until they reach a corresponding update point is a way of mapping control from the old version to the new one. Some solutions allow the programmer to transform the stack in-place [20], while others delay an update until no updated functions are on stack [6, 35].

DSU systems can now dynamically update real releases of large programs. In addition to the applications presented in this paper, Kitsune has updated the Snort IDS, the Tor anonymous router, the Icecast multimedia server [15]. Related systems have updated the Voldemort key-value store, the H2 database, and a Quake 2 port [29]; the PostgreSQL database [20]; and the Minix [12] and Linux [6] operating systems.

2.2 Threats to Availability

If all goes well, DSU provides a better service than the alternative: critical updates can be applied in a timely fashion without disrupting existing sessions and without dropping performance- or safety-critical in-memory state. But there are reasons for things not to go so well.

First off, there could be errors in the new program version. New bugs may escape notice during the testing process and only manifest once a dynamic update is applied, e.g., as a crash or wrong answer. But there is also the problem of bugs in the DSU-specific parts of the program, e.g., due to the placement of the kitsune_ directives, or the specification of the state transformation. Roughly speaking, these can be broken down as *timing errors* and *state transformation errors*.

As an example timing error, suppose that the programmer placed a kitsune_update call inside the code for the put function. As a multithreaded program, our key-value store needs synchronization to avoid data races on table via table_mutex. Suppose that a thread calls put, acquires the mutex, and then reaches kitsune_update before releasing the mutex, and an update is available. The thread will block there, not releasing the mutex. As a result, all other threads will be prevented from acquiring the mutex, which could inhibit quiescence, and lead to other correctness problems.

A state transformation error can arise when the code to transform existing state is wrong. For example, perhaps for our update, the default for field t is mistakenly left uninitialized, rather than explicitly initialized to a default value (like string). Then code that retrieves the updated entries may behave incorrectly. Or, suppose that in the new version the SIZE variable is larger than in the old version. What should happen is that a new, larger table should be allocated, its contents initialized with new entries, and the old entries and table should be freed. But a programmer might mistakenly keep the old table, just updating its entries. As such, the new version may access the table out of bounds, assuming the table is larger than it was transformed to be.

While post-update failures are an extreme threat to system availability, a less extreme threat is the delay in service that occurs while an update is taking place. The delay is due to the time to quiesce the program threads, and the time to transform an arbitrarily large program state. For instance, in the example shown in Figure 2, the state transformation must iteratively update every existing entry in the table. If SIZE is large, this could take a very long time. And yet, the server can perform no useful work while the transformation is taking place.[2]

In the next section, we present MVEDSUA, our approach to addressing these problems.

3 Better DSU with MVEDSUA

MVEDSUA extends a DSU system with *multi-version execution (MVE)*. This section describes MVE and then explains how MVEDSUA employs it to reduce the pause in service due to a dynamic update, and to gracefully tolerate errors that might be introduced by it.

3.1 Multi-Version Execution (MVE)

Multi-Version Execution (MVE) allows several processes to execute in parallel over the same inputs in order to increase either reliability (so that if a bug affects only some of the versions, the others might be able to continue execution) or security (so that attacks would need to succeed in all versions to go undetected, significantly raising the bar for a successful attack). Most MVE systems operate at the system-call level [16,17,19,33,36], checking that all processes issue the same sequence of system calls and ensuring these produce the same results. For instance, if two processes P_1 and P_2 issue a `read` system call from a socket, MVE ensures that the socket is only read once and both processes receive the same data.

A particularly efficient way to perform MVE is to define one of the processes as the *leader* and all others as *followers* [17]. The leader interacts with the underlying operating system (OS) by issuing system calls, while the followers check that their system calls match the leader's, in which case they get their results from it, not from the

[2]Note, however, that even this DSU pause time is lower than serializing in-memory data to disk, stopping, updating, and deserializing on restart, which can be up to 28 seconds for a 10GB Redis heap [15], or 13 seconds for a 1GB H2 heap [29]. DSU also supports data representation changes.

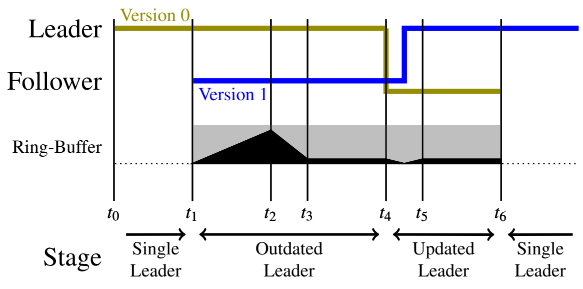

Figure 3: MVEDSUA's update stages.

OS [17,19,36]. This is typically done using a *ring buffer*. The leader registers each system call and its result on the ring buffer, while each follower matches each system call with its current position on the ring buffer, ensuring that it is about to perform the same system call with the same arguments. Then, instead of performing it, followers simply read the leader's results from the ring buffer.

A *divergence* occurs when the sequence of system calls issued by a follower does not match that of the leader. Sometimes a divergence indicates a problem, but not always. For instance, a divergence could occur when the leader and follower are executing different versions of the same program and they issue different, but equivalent, system calls. A proven way to tolerate expected divergences is to provide the MVE system with a set of *rewrite rules* to map the sequence of system calls of the follower into a different sequence that matches the leader [17, 21, 25, 27]. The use of rewrite rules to tolerate divergences following an upgrade is a key element of MVEDSUA, which we describe next.

3.2 MVEDSUA: Combining DSU and MVE

MVEDSUA employs a pleasingly simple combination of DSU and MVE. We illustrate how it works in Figure 3. At time t_0, we deploy a DSU-enabled program executing in a degenerate MVE mode with a single leader and no follower—the **single-leader stage**. This executes the program in a lightweight MVE runtime that will accept another version later while imposing minimal overhead.

When an update becomes available at t_1, MVEDSUA uses the MVE system to create a new follower by forking the leader. Then, MVEDSUA uses the underlying DSU system to perform the dynamic update on the follower. This starts the **outdated leader stage**. In the meantime, the leader keeps providing service, registering its system calls on the ring buffer. If the buffer gets full, the leader blocks until the follower finishes the update.

The follower finishes the update at t_2. At this point it is running the new version while the leader is running the old version. The follower will start consuming the system calls that the leader registered on the ring buffer. As it does so, MVE confirms that it, the new version, is consistent with the old version's behavior. Of course, there are going to be intentional divergences in behavior, e.g., because the new version added new features. These will be handled by a programmer-provided mapping, as discussed in §4. Eventually, the follower will drain the ring buffer, catching up with the leader, at t_3. For the rest of the outdated leader stage, the new version continues to be tested against the old one.

At t_4, the operator decides to expose the updated interface to clients by promoting the updated version and demoting the out-of-date version. This is a fast operation that ends at t_5, and involves the leader registering a special demotion/promotion event on the ring buffer, and becoming a follower immediately, no longer processing incoming events. During this time, there are two followers and no leader providing service. The usage of the ring buffer drops to zero as the new-version follower consumes the remaining system calls.

At t_5, the new version becomes the leader and resumes service. During the **updated leader stage**, the new version registers events on the ring buffer that the outdated follower will now consume and validate. A reverse developer-provided mapping will handle expected divergences in system call sequences. This stage may be bypassed if constructing the reverse mapping is too difficult owing to substantial changes in the new version.

Finally, at t_6, the operator decides that the update was successful and terminates the outdated follower. From this point on, MVEDSUA resumes the single-leader stage.

In sum, MVEDSUA improves the reliability of dynamic updates and the availability of system services by:

Reducing update latency. By performing the dynamic update in the follower in parallel with the execution of the leader (between t_1 and t_2), MVEDSUA avoids any pause in service availability.

Handling in-update errors. If the dynamic update fails prior to t_2, MVEDSUA will terminate the follower and revert to a single-leader stage, allowing the old version to carry on. Nondeterministic failures (e.g., due to unlucky timing) can simply be retried; deterministic failures (e.g., due to state transformation errors) can be retried once the update is fixed.

Handling new-version errors. If the update succeeds without incident, MVE will try to match the new version's execution to the old version's during the updated leader stage. Bugs in the new version, or bugs in the update that are residual (e.g., owing to a buggy state transformation), will manifest as a new-version crash or a divergence. In these cases, the follower is terminated and the old version is allowed to proceed until the bug is fixed so the update can be retried.

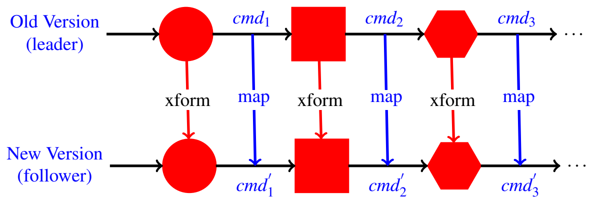

Figure 4: Mapping that ensures old- and new-version states are related by the state transformation.

Handling old-version errors. If the old version crashes but the new version does not, this may indicate an old-version bug fixed in the new version. As such, the new version can be promoted to be the sole leader (jump to t_6) in order to recover. Such a promotion makes sense at any time after t_1. If the old version exhibits a bug between t_5 and t_6 that manifests as a divergence, then it (the follower) will be terminated.

4 Mappings

A key requirement for using MVEDSUA is for the programmer to provide a mapping from system call sequences seen by the leader to an equivalent sequence to be seen by the follower. During the outdated leader stage, old-version events must be mapped to the new version. During the updated leader stage, the situation is reversed.

4.1 Old-version Leader Mappings

After the update is applied, MVEDSUA treats the old version as the leader, using its behavior to confirm that the new version's behavior is reasonable, i.e., that there was not a bug in the update, or in the new code. It does this by confirming that portions of the API that are backward compatible behave the same, from the clients' perspective, before and after the update.

To do this, the programmer should write a mapping that ensures that after processing each client command, both the old version and new version are in *compatible states*. In particular, the new version should be in the same state it would have been in had the old version been dynamically updated at that point. This situation is depicted in Figure 4. The top line depicts the processing of client commands $cmd_1, cmd_2, cmd_3, \ldots$ by the old-version leader. Each of these effects a change on the leader's state, depicted as a red shape, e.g., adding entries to the key-value store. The *map* downarrow indicates the process by which system calls that correspond to these commands are mapped to system calls that correspond to commands $cmd'_1, cmd'_2, cmd'_3, \ldots$ for the new version. Importantly, after each command, the old and new version states should be *related* by the dynamic update's state transformer, here shown with a downarrow labeled *xform*. This means that had the dynamic update occurred after any of these commands instead, it would have produced an MVE system in the same state. As such, a divergence indicates a mistake in either *xform*, in *map*, or in genuine system behavior, e.g., due to a new bug in the new version.

For commands that have the same semantics in the old and new version, it is quite possible that no syscall mapping is needed. For the update introduced in Figure 2, the `GET` and `PUT` commands behave the same in both versions. On the other hand, the new version is likely to have added new features. Our example update introduced two new client-visible commands, `PUT-`*type* and `TYPE`. If we provide no syscall mapping for these commands, the two versions' behavior will diverge. Suppose a client issues the command `PUT-number balance 1001`. This command will be rejected by the old version (since it does not understand `PUT-number`) but accepted by the new version. We could write a mapping that tolerates this difference, but doing so is problematic because the result would violate our expected state relation. In particular, it would no longer be the case that dynamically updating the old version's state would yield the new version's: the new-version follower added a new entry `balance` $\rightarrow$ (`1001`,`number`) to its store, but no corresponding entry in the old version would produce this on an update—the types of all updated values would be `string`. Breaking the state relation will lead to divergences later that may not signal genuine errors. For example, a subsequent command `GET balance` would return an error on the leader, since no mapping is present, but would return a value (`1001`) on the follower. But this divergence does not signal an actual error in the update or the new version; it is simply a to-be-expected difference in behavior.

To avoid this spurious divergence, we can write rules that force the new version to adhere to behaviors defined by the old version. In particular, we can easily write rules to translate new commands that the leader does not understand to commands that the follower doesn't understand either. Rule 1 in Figure 5 encodes this logic in the syntax of Varan's DSL [27]. It says that if a `read` system call receives a string that contains a command with a type component, i.e., of the form `PUT-`*type* or `TYPE`, then it should issue `bad-cmd` to the new-version follower. Since the follower will not understand this command, it will reject it, just as the old version will do for the original.

```
// parse("PUT k1 v1") = ("PUT", NULL, "k1", "v1")
// parse("PUT−string k1 v1") = ("put", string, "k1", "v1")
// parse("TYPE k1") = ("TYPE", NULL, "k1", "") etc.
// Rule 1
read(fd,s,_) {
    (cmd,typ,key,val) = parse($(s))
    return typ != NULL; } as r
                                        => r(fd, "bad−cmd", 7)
// Rule 2
read(fd,s,n) as r => r(fd,s,n) {
                              (cmd,typ,key,val) = parse($(s));
                              $(s) = "$cmd−string $key $val";
                              $(n) += 7; }
```

(a) Updated follower t_2–t_4

```
// Rule 3
read(fd,s,_) {
   (cmd,typ,key,val) = parse($(s))
   return typ == string; } as r
                              => r(fd,s,n) {
                                   (cmd,typ,k,v)=parse($(s));
                                   $(s) = "$cmd $k $v";
                                   $(n) −= 7; }
```

(b) Outdated follower t_5–t_6

Figure 5: Rewrite rules to implement syscall mapping.

If commands understood by the old version are not valid in the new version, sometimes we can write a mapping for these. For example, if the new version dropped support for `PUT`, we could install Rule 2 to translate such commands to `PUT-string`, which should have equivalent semantics. For commands in the old version that simply have no new-version equivalent (such as `TYPE`), we have no choice but to terminate the follower unless the command produces clearly-wrong behavior in the leader, such as a hang or a crash.

4.2 New-version Leader Mappings

During the updated leader stage the new version is the leader so the situation depicted in Figure 4 is slightly different: The new version is on top and the xform arrow is reversed. But the principle is the same: after each command, the two versions should be in related states.

When the new version presents a mostly or fully backward compatible client API, there is little work involved. However, for new commands it may be difficult or impossible to provide a proper mapping. For our example update, `GET` and `PUT` commands will work as usual. But if the client submits a `PUT-`*type* command, there is no complete solution. If *type* is `string`, then we can map the command to a normal PUT command, for which `string` is the default type. This is shown in Rule 3 in Figure 5. For other values of *type*, there is no possible mapping, meaning that the old-version follower will diverge from the leader and be terminated. Up to the point that this happens, MVEDSUA will at least be checking that the new version does not do something obviously wrong, like crash, in which case it can promote the follower. But, in general, the inability to gracefully deal with new-version commands means that updated-leader stage is likely to be less useful at finding update-related errors compared to outdated-leader stage.

5 Implementation Details

Our implementation of MVEDSUA combines the Kitsune [15] DSU system with the Varan [17] MVE system. The bulk of our implementation of MVEDSUA is located inside Varan, in 1202 lines of extra C code. Kitsune required minimal changes, with only 88 lines modified.

The changes to Kitsune support coordinating with Varan to fork a follower and perform the update only there, while aborting the update and resuming normal execution on the leader. This is done by having Kitsune contact MVEDSUA to check if the update should be taken, after stopping all threads at update points. MVEDSUA uses this opportunity to fork execution, aborting the update on the leader and allowing it on the follower. MVEDSUA provides a callback that is invoked on an aborted update, in case some work should be done before resuming; we used this callback for Memcached as discussed in §6.3. One wrinkle arises in the case of multithreaded applications. In modern operating systems, forking a multithreaded program results in only one thread running in the forked process [1]; all other threads must be restarted on the follower. Pleasantly, this is something that Kitsune already does after quiescing at update points, so MVEDSUA piggbacks on that support.

The most significant addition that we made in Varan was efficient support for executing in single-leader mode, which happens for a majority of a MVEDSUA program's lifetime. Normally, the leader logs its intercepted system calls on the ring buffer, and the follower reads from the buffer to match its own intercepted calls. In addition, Varan tracks kernel state that is relevant to MVE, such as a logical process PID (used for both leader and follower), event-poll descriptors, and more. In single-leader mode, the ring buffer is not used, but system calls must still be intercepted to track kernel state. This state is then used when forking into leader-follower mode later on. The overhead due to syscall interception is relatively small, as Varan does it via binary rewriting [17].

Table 1: Mvedsua rewrite rules per Vsftpd pair.

Versions	# rules	Versions	# rules
1.1.0 → 1.1.1	0	2.0.0 → 2.0.1	0
1.1.1 → 1.1.2	2	2.0.1 → 2.0.2	1
1.1.2 → 1.1.3	0	2.0.2 → 2.0.3	1
1.1.3 → 1.2.0	2	2.0.3 → 2.0.4	1
1.2.0 → 1.2.1	0	2.0.4 → 2.0.5	1
1.2.1 → 1.2.2	0	2.0.5 → 2.0.6	0
1.2.2 → 2.0.0	3	Average	0.85

```
read(_,_,_), write(_,"500 Unknown command",_)
    => read(_,"FOOBAR\r\n",8),
       write(_,"500 Unknown command",_)
```

Figure 6: Rewrite rule for Vsftpd to safely redirect unknown commands to newer version.

6 Case Studies

We tested MVEDSUA by using it to perform multiple dynamic updates to three high-performance servers: Vsftpd, Redis, and Memcached. We found that few DSL rules were needed, and only Memcached required nontrivial code changes to work with MVEDSUA.

6.1 Vsftpd

Vsftpd is an open-source FTP server and is a useful benchmark because several other DSU systems have used it for evaluation [15, 20]. We used 14 versions (1.1.0–2.0.6) tested in 13 pairs which cover three years of releases. On average, we found we needed only one DSL rule per update, on average. Table 1 reports the number of rules required (the same number for both the outdated and updated leader stages, where the latter were easily derived from the former).

One interesting case was version 1.2.0 introducing a new command, STOU, which stores a unique file in the current working directory. So, following the methodology in §4.1, we use a rule to trigger an invalid command on the new version while the old version is the leader. Figure 6 shows a general form of this rule: when the old version reports a 500 (unknown command), it rewrites the STOU command to another invalid command, guaranteed to result in the same behavior in the new version.

Interestingly, when the new version is the leader a new command would normally cause an irreparable divergence (per §4.2). However, in this case, both the old and new versions share the same file system, so the effect of STOU executed on the leader will be visible to subsequent commands, such as GET, issued to both leader and follower. As such, we can write a rule to tolerate the divergence and the two versions will remain in sync.

6.2 Redis

Redis [4] is a single-threaded, in-memory key-value store, typically used as a database cache or a message broker, with the option of persisting the store contents to disk. We used versions 2.0.0, 2.0.1, 2.0.2, and 2.0.3, which were also used to evaluate Kitsune [15].

The Redis versions used with Kitsune needed only few bug fixes and changes needed to build those versions with recent versions of GCC and libc. We updated 7 lines per version (the same lines in the same files), and wrote a DSL rule for 2.0.0 $\rightarrow$ 2.0.1 as 2.0.1 reverses the order of two system calls when handling client commands.

6.3 Memcached/LibEvent

Memcached [3] is a multi-threaded, in-memory key-value store, typically used for caching results of database queries, API calls, or page rendering. We used versions 1.2.2, 1.2.3, and 1.2.4, which were also used to evaluate Kitsune [15]. Memcached is built around LibEvent [2], which replaces the event loop found in many applications. With LibEvent, applications register file descriptors, timeouts, and signals they are interested in; and function pointers to execute when these events happen. LibEvent's internal event loop invokes the appropriate callbacks when events occur.

Memcached required changes to work with MVEDSUA so that it could properly abort the update on the leader. One change involved writing a callback to reset some of LibEvent's state (see §5), to avoid spurious divergences due to the order that events are handled. In particular, each Memcached thread registers many events in which it is interested. When several events become available, LibEvent invokes the callbacks in a round-robin fashion, remembering where it was after each invocation. The updated follower does not have this memory, which means it may handle events in a different order, causing spurious divergences. Resetting the state on the leader ensures that it and the follower are in sync.

A more interesting problem arose because of the use of LibEvent. Kitsune-enabled Memcached works by placing an update point just after the call to the LibEvent event processing loop. This processing loop would normally exit when an update is signaled, and thus control would reach the update point. Once all threads had done so, Kitsune would normally update the code and cause each thread to unwind its stack and reinvoke its entry point (see §2.1). With MVEDSUA this will happen on the follower and work fine, but on the leader the update is aborted. If execution is just allowed to continue, the leader will terminate, wrongly. We could solve this problem by having the abort callback do the same stack unwinding and reinvocation for the original version that the follower does for the new version. But it turns out this is not sufficient: When trying to demote the leader later no update would be available to cause the program to exit its event loop. As such, we extended Kitsune to optionally consider instances of system call `epoll_wait` as an update point, as indicated by a flag set in the application. This allows LibEvent to reach an update point frequently, in a loop, without exiting. As such, it works for establishing quiescence when updating originally, and when doing so to swap leader and follower.

In total, we modified 114 lines for each Memcached version (same lines in the same files). No version changed the sequence of system calls or added any commands, so we did not write any DSL rules.

7 Experimental Evaluation

Here we describe how we used the applications described in §6 to evaluate the performance and efficacy of MVEDSUA. In summary, we found that MVEDSUA introduces 3-9% overhead during the single-leader stage, which spans the vast majority of program execution, and 25-52% overhead when monitoring an update; that MVEDSUA eliminates update pauses completely; and that it recovers from real and realistic update errors.

7.1 Performance

We evaluated MVEDSUA's performance by measuring its effect on the throughput of our test applications. We ran Redis versions 2.0.0 and 2.0.1, and Memcached versions 1.2.2 and 1.2.3, both with the benchmark Memtier version 1.2.10. We ran it for 6 minutes, starting from an empty store, and using a 90% read 10% write workload. For Vsftpd we used versions 2.0.5 and 2.0.6 with a custom benchmark script which simply logs in and repeatedly downloads a particular file for 60 seconds before logging out. We considered a "small" version of the benchmark with a 5B file, and a "large" version with a 10MB file, with the former stressing user-space FTP command processing, and the latter stressing the kernel-space syscall processing, which puts a load on Varan.

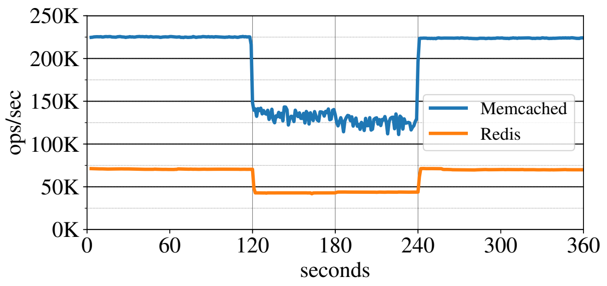

Figure 7: Performance while updating Memcached and Redis with MVEDSUA during all update stages.

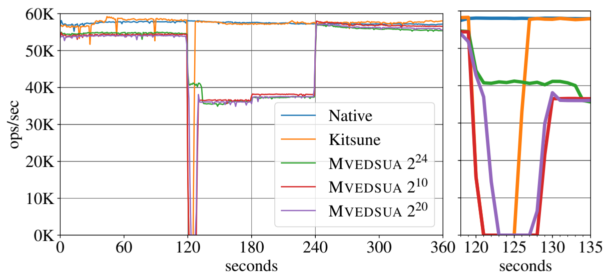

Figure 8: Performance while updating Redis with a large program state and different buffer sizes. The right-hand side zooms-in on the 15 seconds following the update.

We performed this evaluation on a machine equipped with two Intel(R) Xeon(R) E5-2450 CPUs, each with 8 physical cores; and 192GB RAM. To prevent NUMA memory-access noise, the server processes execute on one CPU and the client benchmark on the other. All live threads have a dedicated core. All results report the average and standard deviation of 10 runs.

Unless otherwise specified, Varan was configured to use a buffer size of 256 entries. Each entry in the ring buffer is 32B long; the largest buffer used with 2^{24} entries requires 512MB of memory. We note that our performance results reflect a worst-case scenario, with the client on the same machine as the server. A more realistic scenario, with the client on a different machine in a different location would incur a lower performance overhead as measured by the client benchmark, since network latency would hide some of MVEDSUA's overhead.

Steady-State. We evaluated MVEDSUA's performance and that of related configurations; results are in Table 2. The table shows MVEDSUA's performance in its two key modes of operation: single leader mode (MVEDSUA-1) and outdated/updated leader mode (MVEDSUA-2). The former is a program's normal mode of operation, and its overhead is small compared to a *Native* binary: 3%–9%. The latter mode imposes around 50% overhead, but is only enabled for a relatively short period: just during an update and afterward, while testing it. Roughly these overheads correspond to the component overheads of *Kitsune* (only) added to single-leader Varan (Varan-1) or leader-follower Varan (Varan-2), respectively.

Update time. To understand how well MVEDSUA masks the pause due to updating, we evaluated the performance of MVEDSUA when performing an update on Redis 2.0.0 → 2.0.1 and Memcached 1.2.2 → 1.2.3.[3] In our experiments, the update happens at 120 seconds (t_1 in Figure 3); the promotion/demotion (t_4) happens at 180 seconds; and single leader mode resumes (t_6) at 240 seconds. Figure 7 shows how many operations Memtier completed per second, on average. Times t_1 and t_6 are immediately visible, with MVEDSUA entering and exiting MVE. The performance levels match MVEDSUA-1 and MVEDSUA-2 in Table 2. For Redis, t_4 is visible as a slight increase in throughput. A key takeaway is that service never stops during the updating process.

To see the impact of a larger state on update time, we modified the Redis experiment to initialize the store with 1M entries before the benchmark starts, which results in a resident process size of around 250MB. We then repeated the experiment for native without an update, Kitsune performing 2.0.0. → 2.0.1, and MVEDSUA performing the same update with three ring buffer sizes: 2^{10}, 2^{20}, and 2^{24} entries.

Figure 8 shows the results. We measured the pause introduced by each update as the maximum latency reported by Memtier throughout all 10 runs: 6,200ms for Kitsune, 8,900ms for MVEDSUA-2^{10}, 6,900ms for MVEDSUA-2^{20}, and 260ms for MVEDSUA-2^{24}. The native maximum latency is 300ms. The results thus show that, with a large enough ring buffer, MVEDSUA can mask the update pause introduced by DSU.

Note that executing in outdated leader mode is important for masking update latency. This is because the ring buffer can be drained in parallel with providing service, as opposed to drained while service is paused, prior to switching to the new version. To confirm this we configured MVEDSUA to promote the updated version and terminate the outdated version immediately, measuring the maximum latency throughout the process as 3,000ms, as compared to at most 300ms when running for a time in outdated leader mode.[4]

[3]Vsftpd is essentially stateless, which means its update-time pause is already low, so we did not measure it.

[4]The follower will take 6,200ms to do the update, during which time the leader is filling the ring buffer. When the update switches to

Table 2: Steady-state performance and overhead of Memcached, Redis, and Vsftpd.

Version	Memcached		Redis		Vsftpd small		Vsftpd large	
	Ops/sec ×1000	Over head	Ops/sec ×1000	Over head	Ops/sec	Over head	Ops/sec	Over head
Native	249 ± 1.05	—	73 ± 0.46	—	2667 ± 5.53	—	118 ± 0.06	—
Kitsune	242 ± 1.52	3%	74 ± 0.31	-1%	2535 ± 5.27	5%	117 ± 0.13	2%
Varan-1	234 ± 0.78	6%	68 ± 0.92	8%	2594 ± 3.14	3%	117 ± 0.10	2%
MVEDSUA-1	226 ± 1.69	9%	69 ± 0.12	6%	2458 ± 4.26	8%	116 ± 0.11	3%
Varan-2	125 ± 2.13	50%	41 ± 0.46	44%	2048 ± 2.71	24%	90 ± 0.18	25%
MVEDSUA-2	121 ± 5.22	52%	43 ± 0.11	42%	2001 ± 4.30	25%	89 ± 0.19	25%

7.2 Fault Tolerance

In this set of experiments, we demonstrate how MVEDSUA is able to detect and recover from a variety of errors.

Error in the New Code. In Redis, revision `7fb16bac` introduces an error that crashes the server when invoking command `HMGET` on the wrong type of data. The error is present in all versions of Redis we tested, and is only fixed on a later release. As an experiment, we undid the error in version 2.0.0, so that the update 2.0.0 $\rightarrow$ 2.0.1 introduces it. Following a dynamic update with Kitsune, the program crashes when a client sends a bad `HMGET` command. But when using MVEDSUA for the update, sending a bad `HMGET` command results in the follower failing, at which point execution reverts to single-leader mode. Clients proceed without incident.

Error in the State Transformation. A Kitsune-developed update had a latent bug: it would free memory still in use by LibEvent. This error could cause a crash if the freed memory were later re-used by the memory allocator. We observed that this error seemed to manifest only when a sufficiently large number of clients were connected to Memcached. With Kitsune, the result was an immediate crash. With MVEDSUA, however, this new-version crash is tolerated, and execution continues with the old-version leader without clients noticing.

Timing Error. Recall that we needed to change Memcached to reset LibEvent's state in the leader after an aborted update (see §6.3). Failure to do so constitutes a kind of timing error in the dynamic update, and leaving out our change may produce a divergence detected by MVEDSUA. While we ultimately fixed the error by adding code to reset LibEvent's state, the fact that divergences are aborted means we could have simply retried the update. In an experiment, we configured MVEDSUA to retry the update after waiting 500ms, and found that the update was always installed eventually, after a maximum of 8 retries with a median of 2 retries.

the follower, it will take half that time to consume the buffer.

8 Related Work

MUC [31] is the first work we know of that combined DSU and MVE. It aims to support an incremental upgrade across a distributed system, for both clients and servers. While MUC is performing an upgrade, old/new clients/servers execute under MVE. As with MVEDSUA, an update starts by forking the current process and then updating the child. Both processes are shepherded by a coordinator that monitors system calls between the two (via `ptrace`) and compares their outputs. Old clients interact with old servers, new clients with new servers, and client upgrades force a switch. Unfortunately, MUC's MVE solution is very limited. First, it runs both processes in lock-step, which means that MUC cannot hide update-induced pauses, and incurs significant overhead on syscall-heavy workloads. Second, MUC cannot handle failures during or after an update due to new-version or update-induced bugs. MUC can handle expected divergences in behavior, but only by annotating system calls in the source code that are expected to diverge. MUC cannot keep states related across versions, in the manner of §4, and has no good way to fix this issue.

UpStart [5] also considers incremental updates to a distributed system, focusing on modular updates to individual nodes. Each node keeps a single object, in one version, and uses simulation objects, one for each version supported but not installed (past or future). Similarly to MVEDSUA, UpStart needs to map operations between different versions of the same object. When such a mapping is impossible to specify, UpStart disallows calls that would expose a divergence. This approach is similar to MVEDSUA's: New behaviors are disallowed to propagate to the follower in the outdated leader stage; such behaviors would expose a divergence in the updated leader stage, which result in terminating the follower. UpStart does not tolerate update errors.

POLUS [8] also supports incremental updates, but within a process rather than across a distributed system. It allows multiple threads to run different code versions so they can be updated one at a time. It employs transformation functions to map shared data to a view consistent with the accessing thread's version. POLUS does not support rollback on error and does not support updates with incompatible backward mappings. For example, for our update in Figure 2, POLUS could not back-transform store entries with non-string type.

TTST [10] proposes an approach for validating state transformations based on process-level updates. TTST first updates the old version (running in a process *Old*) to the new version (running in a process *New*) using forward state transformations, and then updates the new version to an old version (running in a process *Reversed*) using backward state transformations. It then compares *Old* and *Reversed* in order to detect potential state transformation bugs, which would cancel the update. MVEDSUA is more general than TTST in that it may find state transformation errors that escape TTST (e.g., when both the forward and the backward transformations are wrong, but in a reversible way, or when the mistake manifests after update-time); can find other types of bugs introduced by the update process, particularly bugs introduced by the patch itself; and MVEDSUA can mask the pause times introduced by live updates, which TTST cannot (more precisely, TTST adds 0.1–1.2s to the update time). Note that TTST could also benefit from the overall MVEDSUA approach, by performing the forward and backward updates in the background.

Proteos [12] provides OS support to update a set of processes atomically. Similarly to MVEDSUA, if the update fails, Proteos simply rolls it back, allowing the to-be updated processes to resume execution in the old version. However, Proteos stops all processes while the update is taking place, pausing service. Furthermore, Proteos does not monitor the updated processes after the update is completed, so it cannot detect post-update errors.

MX [16] uses MVE to run two program versions in parallel and tolerate errors in one by using the other. MX performs a checkpoint of each version at each system call. With MX, if version V_F fails but version V_{OK} does not, MX reverts V_F to the previous checkpoint, patches V_F with the code of V_{OK} and executes it until the next system call, and then, if successful, reverts back to the original code of V_F. MX can tolerate failures introduced in the new version, but it cannot add a new version in mid-execution, so it is not suitable for DSU. Also, MX executes versions in lock-step, synchronizing at each system call, so it incurs a significant overhead. Finally, MX does not tolerate any system call divergence, which restricts its ability to deploy release-level versions.

Imago [9] can update large distributed systems by launching a full copy of the system under upgrade. Imago treats the whole system as a black-box, intercepting the inputs (e.g., HTTP requests) to send them to both versions, and comparing the outputs (e.g., database queries). If an update fails, Imago can terminate the duplicate system and revert to the outdated version without any client-visible disruption. Imago can also detect semantic errors by comparing outputs at the database level, with programmer-specified data conversion logic to tolerate divergences between the two versions. Imago requires a mostly stateless system which keeps its state on a data store that can be shared with other systems (and whose semantics cannot change). MVEDSUA does not require a particular architecture to be effective, supports expressive updates (including representation changes), can work with any updatable system (and adapted to many DSU systems) and with memory-only stateful applications. Both Imago and MVEDSUA tolerate update errors by using more computational resources, but MVEDSUA operates at a lower level and requires less resources than Imago.

An approach to deal with long update times is to perform *parallel state transformation* using several threads [29, 34] or *on-demand (lazy) state transformation* [22, 26, 29], transforming data as it is accessed after the update. Unfortunately, lazy transformation is particularly challenging for C programs, which can easily break proxies and other abstractions used to support it. Parallel transformation can reduce the pause but not eliminate it.

9 Conclusion

Dynamic software updating (DSU) can be an effective solution to the problem of updating stateful applications without disrupting service. However, DSU systems introduce pauses during updates and require programmer assistance which is prone to introducing errors. In addition, software updates themselves can introduce errors which can escape off-line testing. MVEDSUA is a novel system that combines Kitsune, a modern DSU system, with Varan, a modern multi-version execution system, to deliver a solution that both masks the pauses introduced by DSU systems and tolerates a variety of failed updates. We evaluated MVEDSUA on several high-performance servers—Redis, Memcached and Vsftpd—and found that it imposes low overhead in steady state, masks the update-time pauses, and tolerates real and realistic update errors. We believe MVEDSUA's ideas should apply to other DSU systems. An interesting future direction is to consider automated inference of syscall mappings based on test data or multi-version static analysis.

References

[1] fork — linux programmer's manual. `http://man7.org/linux/man-pages/man2/fork.2.html`. Accessed: 2018-05-02.

[2] libevent — an event notification library. `http://libevent.org/`. Accessed: 2018-05-02.

[3] memcached: a distributed memory caching system. `https://memcached.org`. Accessed: 2018-05-02.

[4] Redis homepage. `https://redis.io/`. Accessed: 2018-05-02.

[5] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[6] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the 4th European Conference on Computer Systems (EuroSys'09)*, Mar.-Apr. 2009.

[7] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, Jul 2001.

[8] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *Proc. of the 29th International Conference on Software Engineering (ICSE'07)*, May 2007.

[9] T. Dumitraş 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 ACM/IFIP/USENIX International Conference on Middleware (Middleware'09)*, Nov. 2009.

[10] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *Proc. of the 27th USENIX Conference on System Administration (LISA'13)*, Nov. 2013.

[11] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum. Mutable Checkpoint-Restart: Automating Live Update for Generic Server Programs. In *Proc. of the 15th ACM/IFIP/USENIX International Conference on Middleware (Middleware'14)*, Dec. 2014.

[12] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, Mar. 2013.

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

[14] C. Hayden, E. Smith, E. Hardisty, M. Hicks, and J. Foster. Evaluating dynamic software update safety using efficient systematic testing. *IEEE TSE*, 2012.

[15] 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 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12)*, Oct. 2012.

[16] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[17] P. Hosek and C. Cadar. Varan the Unbelievable: An efficient N-version execution framework. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Mar. 2015.

[18] M. Kerrisk. LCE: Checkpoint/restore in user space: are we there yet? `https://lwn.net/Articles/525675/`, 2012.

[19] K. Koning, H. Bos, and C. Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proc. of the 2016 46th International Conference on Dependable Systems and Networks (DSN'16)*, June 2016.

[20] K. Makris and R. A. Bazi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the 2009 USENIX Annual Technical Conference (USENIX ATC'09)*, June 2009.

[21] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)*, Aug. 2012.

[22] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proc.*

of the Conference on Programing Language Design and Implementation (PLDI'06), June 2006.

[23] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *USENIX NSDI*, 2013.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, Dec. 2002.

[25] L. Pina, A. Andronidis, and C. Cadar. FreeDA: Incompatible stock dynamic analyses in production. In *Proc. of the 2018 ACM International Conference on Computing Frontiers (CF'18)*, May 2018.

[26] L. Pina and J. Cachopo. Atomic dynamic upgrades using software transactional memory. In *Proc. of the 4th Workshop on Hot Topics in Software Upgrades (HotSWUp'12)*, June 2012.

[27] L. Pina, D. Grumberg, A. Andronidis, and C. Cadar. A DSL approach to reconcile equivalent divergent program executions. In *Proc. of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, July 2017.

[28] L. Pina and M. Hicks. Tedsuto: a general framework for testing dynamic software updates. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'16)*, Apr. 2016.

[29] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *Proc. of the 29th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'14)*, Oct. 2014.

[30] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, Jan. 1995.

[31] W. Qiang, F. Chen, L. T. Yang, and H. Jin. MUC: Updating cloud applications dynamically via multi-version execution. *Future Generation Computer Systems*, 2015.

[32] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proc. of the 6th International Workshop on Automated Debugging (AADEBUG'05)*, Sept. 2005.

[33] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the 4th European Conference on Computer Systems (EuroSys'09)*, Mar.-Apr. 2009.

[34] K. Saur, M. Hicks, and J. S. Foster. C-strider: Type-aware heap traversal for C. *Software, Practice, and Experience*, May 2015.

[35] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: A VM-centric approach. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'09)*, June 2009.

[36] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. D. Sutter, and M. Franz. Secure and efficient application monitoring and replication. In *Proc. of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*, June 2016.

[37] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, 2010.

[38] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, Sept. 2011.