Abstract
Dynamic Software Updating (DSU) is a general-purpose technique for patching stateful software without shutting it down, which enables both timely updates and non-stop service. However, applying an 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**, a system that solves these problems by augmenting a DSU system with support for Multi-Version Execution (MVE). To start, **MVEDSUA** performs an update in parallel with the original system, thereby avoiding any service delay. Then, it monitors that the updated and original systems’ responses agree when given the same inputs. Expected differences are 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 and Varan, state-of-the-art DSU and MVE systems respectively, 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-related 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 system results in an unacceptable loss of availability, so a more sophisticated, “rebootless” updating technique must be used. One common approach is to perform a **rolling upgrade** [12]. This technique supports updating stateless nodes in a distributed service (e.g., application servers in a web service) by gracefully directing new connections away from a node and then stopping, patching, and restarting it once its work queue is empty. Thus each node is upgraded (in a “rolling” fashion) without disrupting the overall service.

Rolling upgrades work in many cases, but not always. Nodes with long-running sessions (e.g., SSH and remote access servers) present problems because they cannot be updated until sessions terminate. This is because sessions are **stateful**, and dropping a session state ungracefully could be disruptive. Stateful servers are problematic in general. For example, the Snort intrusion detection system [43] builds a substantial in-memory state machine to detect multi-packet attacks. Shutting down and restarting Snort will drop this state machine and thus potentially miss a mounting attack [24]. Servers that rely on (large) caches for good performance are also problematic. For example, a typical database management system has a large cache; dropping this cache can degrade performance significantly at the time of the update. In-memory databases like Redis and Memcached would require checkpointing state before shutdown and restart, which can be time consuming. The 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, “so that the data can remain live across a software upgrade and thereby minimize disruption” [34]. In short: While rolling upgrades work well on stateless servers, they don’t solve the problem of updating the stateful components of a service.

**Dynamic software updating** (DSU) is a technique for updating stateful servers without shutting them down. 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. DSU technology is available in commercial products, e.g., for patching Linux [9], Java VM-based applications [29, 35], telecom systems using Erlang [8], and even satellite systems [47]. The research community has pushed the envelope further, developing support for full release-level updates to substantial applications, including operating systems, data management systems, and various servers [13, 18–20, 25, 33, 41, 46].

---

**MVEDSUA** (“MVE” + “DSU”) is pronounced “Medusa”.

---

**Authors:**
- Luís Pina, George Mason University, lpina2@gmu.edu
- Anastasios Andronidis, Imperial College London, a.andronidis15@imperial.ac.uk
- Michael Hicks, University of Maryland, mwh@cs.umd.edu
- Cristian Cadar, Imperial College London, c.cadar@imperial.ac.uk
Dynamic updates will improve a system while keeping it running, but only if nothing goes wrong. What could happen? The update itself may have errors—many patches that aim to fix bugs end up introducing new ones [52]. Or the update may be correct, but the mechanics of applying it at run-time may be the issue. 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 may have bugs that cause the program to fail [22, 40]. Even if transformations are correct, they might take a long time to perform if the state to transform is very large (e.g., the whole heap), temporarily halting service.

As a solution to these problems, this paper presents VEDSUA, a novel DSU system that makes use of multi-version execution (MVE) technology [11, 14, 15, 27, 32, 44, 49, 51]. An MVE system works by running multiple instances of a program at once, ensuring that each sees the same inputs and confirming that all produce the same (or equivalent) outputs. When a dynamic update becomes available, VEDSUA forks the current running program, called the leader, and initiates the update on the other copy, called the follower. While the update takes place, the leader continues to process requests, avoiding any long, update-related pause, e.g., due to state transformation. Once the update is complete, VEDSUA’s MVE component monitors the behavior of both versions. Any disagreement may signal a bug in either the new version or the update, so VEDSUA terminates the follower, effectively rolling back the update. Importantly, VEDSUA’s use of MVE ensures that no state changes made after the update are lost. The alternative of simply falling back to a (possibly checkpointed) instance of the old-version program would mean losing such state, since that version would not have been kept up to date by MVE. After running for a while with the old version as leader, VEDSUA can promote the new version to leader, and eventually drop the old (demoted) follower.

VEDSUA’s novel use of MVE creates some challenges. In particular, the behavior of the old and new versions should change in most cases, due to added features and bug fixes. Yet MVE judges any divergence in external behavior as problematic. There are two situations to consider. The first is when the new behavior is superficially different from the old, but both are equivalent to the client. For example, a single system call in the old version might be broken into multiple system calls in the new version. VEDSUA allows the programmer to specify such allowed differences using domain-specific languages (DSLs) provided by modern MVE systems [28, 32, 38].

The second, more interesting situation is when the new version’s behavior purposely disagrees with the old. For example, the new version might support a new client command. For this version, executing the command updates the server’s state, while the old version rejects the command and does not change the state. As such, tolerating the syscall differences will result in later divergences. To handle this situation, we use the MVE DSL to specify that semantically dissonant system call sequences leave the leader and follower in equivalent states. For the case of the added command, the DSL can be used to direct an invalid command to the (updated) follower so its subsequent behavior, and state, matches that of the (outdated) leader. As a result, VEDSUA essentially enforces the semantics of the old version while it is the leader, testing that the new version matches that semantics when it should. Once operators are confident the new version is behaving correctly, they make it the leader and thus expose (and test) its new semantics.

We implemented VEDSUA by extending Kitsune [25], a DSU system for C programs, with MVE support from Varan [28], a modern, high-performance MVE system. Our experience using VEDSUA has been promising. We used it to perform multiple dynamic updates on three high-performance servers: Memcached (2 updates), Redis (3), and Vsftpd (13). The added programmer effort of using VEDSUA was modest: No DSL rules were needed for either Memcached update, one was needed for Redis, and, on average, one was needed for each Vsftpd update. Only Memcached required interesting code changes (about 100 lines) to work properly with VEDSUA, owing to its use of multiple threads and stateful libraries. In terms of run-time overhead, we found that VEDSUA imposes 3–9% overhead on throughput during normal operation, as compared to 0–3% for Kitsune alone. While VEDSUA is monitoring both the original and updated versions, the overhead is 25–52% (which essentially matches Varan’s overhead). Finally, we observed that VEDSUA can completely eliminate the pause due to updating and that it can detect and recover from a variety of errors, including those due to failed update specifications, and failures in the updated code itself.

Compared to existing solutions, VEDSUA is the first system that simultaneously (1) performs dynamic software updates, (2) hides the update latency, and (3) tolerates errors due to bugs in the new version or the update process. Mx [27] uses MVE to run two program versions in parallel and tolerate errors in one by using the other. However, Mx does not support DSU (it can run two versions with MVE from the beginning, but it cannot add a new version in mid-execution), it does not tolerate any system-call divergence, and it introduces a performance overhead of up to 16x. As VEDSUA, Proteos [20] can roll back to the old version when an update fails, but it does not monitor the updated processes after the update is completed, so it cannot detect post-update errors such as those due to buggy patches. Furthermore, Proteos cannot tolerate update-induced delays as it pauses the application until the update completes. MUC [42] combines
2. DSU: Background and Problem Statement

This section sets up the problem we are trying to solve: How to quickly and reliably update a long-running, stateful service. We introduce a running example update, used throughout the paper. We then explain how standard techniques such as rolling upgrades struggle with this example. Next, we introduce Dynamic Software Updating (DSU) as a solution that can handle general-purpose upgrades to stateful services, including our example. Finally, we outline key challenges faced by the state-of-the-art DSU systems, which MVEDSUA addresses.

2.1 Update Example

Figure 1a shows the API of a key-value store. The store is contained in the global structure table, which has SIZE entries. Each entry has two fields: key and val. Clients store and access data by calling functions put and get, respectively. The application server exposes this API through a simple wire protocol in the form of text commands, such as PUT balance 1000 and GET balance, which the server (respectively) translates to calls to put and get.

Figure 1b shows an update to this program that: (1) extends entry with a t field that indicates the value's type; and (2) defines some standard types string, number, and date. This change impacts the signature of put, which now takes the typ as an argument. The server assigns type string to outdated client requests (e.g., PUT balance 100 translates to put("balance", "100", string)). Updated requests can specify the type explicitly (e.g., PUT-number balance 100 translates to put("balance", "100", number)). The request GET balance does not change. A new request form TYPE balance translates to type("balance"), returning the key’s type.

2.2 Challenges of Stateful Upgrades

Suppose we would like to upgrade a running version of the key-value store according to the above change. The simplest way to do so is to stop the old version and restart with the new one. The problem is that our server is stateful, due to its maintenance of state. Stopping the old version drops this important state, harming clients. For example, a stop-restart after client request PUT balance 1000 would cause subsequent request GET balance to fail, rather than return 1000 as expected.

The industry-standard rolling upgrade [12] approach does not immediately help us here. Rolling upgrades work by shutting down and restarting individual nodes when they have completed their work, relying on the other nodes to maintain the overall service. Ultimately, individual nodes must be restarted, and if these are stateful, that state will be lost.

To mitigate this problem, a server could checkpoint its state to persistent storage on exit, and restore it when starting in the new version. This process presents some challenges. First, the state format may change between versions, as happens in our example, necessitating a backward-compatible checkpoint/restore protocol. While our example can be handled easily (default a missing type field to string), a series of changes becomes more difficult to handle, especially in highly optimized systems (e.g., to data structures in key-value stores like Redis or MongoDB). Second, the time required to persist and restore the state can be non-trivial, and therefore lengthen the latency of the full-service update. For example, checkpointing and restarting a 10GB Redis heap took 28 seconds in one experiment [25], and 1GB H2 heap took 13 seconds in another [39]. This delay is sufficiently

```c
typedef char str;
typedef int typ;
#define SIZE 1024

struct entry { str key; void* val; typ t; }
struct entry* table[SIZE];

void put(str key, void* val); typ t; }

(a) Original
(b) Update

Figure 1: An update for an in-memory key-value store.

DSU with MVE to support incremental upgrades across a distributed system, ensuring that old clients interact with old servers, new clients with new servers, and client upgrades force a switch. MUC’s approach is very limited: (1) it cannot tolerate update-induced pauses as it runs both processes in lock-step; (2) it introduces 23–87% steady-state performance overhead; (3) it cannot handle failures during or after an update; and (4) it cannot keep states related across versions, in the manner of §3.3, and has no good way to fix this issue. §7 provides a detailed comparison of MVEDSUA with prior systems.

In summary, the main contributions of this paper are:

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

(2) A prototype implementation of MVEDSUA, using Kitsune [25] and Varan [28], together with an empirical evaluation on the high-performance servers, Redis, Memcached and Vsfptd. Our evaluation on more than a dozen updates shows that MVEDSUA can effectively reduce update latency while incurring only a modest overhead of 3–9% in steady state. For these systems, MVEDSUA can also correctly detect, abort, and recover from erroneous updates; and can tolerate expected divergences in behavior across versions.
disruptive that Facebook avoids it on updates to its Memcached nodes by customizing Memcached to store its state on a ramdisk that the new version can immediately connect to [17]. Of course, Facebook’s approach only works if the internal representation of the state does not change between versions; this is not true for our example.

A related mitigation to state checkpointing is state replication. Most distributed services employ a protocol for state replication to ensure fault tolerance. If the state representation does not change due to an update, this replication layer is sufficient to preserve a node’s state during a rolling upgrade—the restarted node will warm its state from nearby replicas. But this approach suffers similar problems to explicit persistence. Changes in state representation necessitate a change to the replication protocol that is compatible across versions. Even if this change is made, warming a replica post-upgrade can take a while, hurting performance in the meantime. Again, this motivates Facebook’s fast, in-memory approach for Memcached.

Finally, we observe that not all stateful software services that might benefit from on-the-fly replication necessarily can take advantage of distribution and replication. Satellites must provide non-stop service and adapt to changing technologies during their long lifetime, which pushes satellite control systems to feature DSU [47]. The Internet-Of-Things (IOT) promises a plethora of devices that must be updated due to security concerns and changing requirements. Updating such IOT devices is an open problem, and DSU will be unavoidable [48]. Non-Volatile Memory (NVM) makes all program state transparently persistent. As a consequence, the stop-restart approach simply does not work as a valid means of updating the program as it will not reset the state. Again, DSU is central to support evolving programs in NVM deployments [10].

2.3 Dynamic Software Updating

Dynamic Software Updating (DSU) is a solution to the problem of upgrading a running, stateful process. With it, critical updates can be applied in a timely fashion without disrupting existing sessions and without dropping performance- or safety-critical in-memory state, even when that state’s representation changes. Furthermore, DSU can be used as mechanism for per-node updates within an overall rolling upgrade.

DSU systems usually perform updates in two steps. First, they dynamically load new code, which constitutes the logical modifications to the program. Second, they transform the program’s execution state into a form that is compatible with the updated code. In our example, all entries in table before the update must be updated to have an additional t field. Often times, automated support assists with this process (e.g., to reallocate entries that need more space), but the programmer also gets involved. In our example, the programmer might indicate that all existing entries should have t set to string. Control state may also change between versions. For example, new variables or function parameters might be allocated on the stack. Mapping the running program’s control state to one compatible with new code also often requires programmer assistance. For instance, if function put is active at the time of an update, programmers may transform that stack frame by adding a new argument to represent the type of the entry being added to the store.

The task of writing data and control state transformations is fairly simple when a DSU system permits restricting when updates may happen. For example, a DSU system may simply disallow updates when functions modified by the update are active (e.g., put) [31, 33, 46]. Automatic techniques such as this activeness-checking work in some limited cases [9, 35] but, unfortunately, are incomplete in general and may still lead to update errors simply due to performing the update at the wrong time [22, 40]. Some solutions allow the programmer to transform the stack in-place [31], others require the programmer to list when updates can happen [25, 26, 33, 41] or cannot happen [46]. In the former case, programmers can specify update points at which all active threads must pause (or “quiesce”) before the update is applied, with the goal of ensuring all expected state invariants hold prior to transformation, and avoiding races while it takes place.

Following decades of research, state-of-the-art DSU systems largely manage to support full-featured software upgrades while imposing minimal performance overhead, and requiring little extra programmer work [25, 41, 50]. To make a system DSU-ready imposes a modest, one-time cost but little maintenance work. For instance, Kitsune [25] is able to update the Tor anonymous router (76K LoC) with just 159 LoC of changes, and the multimedia server IceCast (16K LoC) with 134 LoC of changes. In both cases combined, support for 18 versions required only a combined effort of 24 LoC. Rubah [41] showed similar numbers for Java. Both of these systems impose only a few percent overhead on normal execution. Besides these examples, state-of-the-art DSU tools have demonstrated updates to the Snort IDS, a Quake 2 port, the PostgreSQL database [31], and the Minix [20] and Linux [9] operating systems, among other examples.

2.4 Threats to Availability

The advantages of DSU hold if all goes well, but there is a potential for problems. 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. There is also the problem of bugs in the DSU-specific parts of the program, e.g., due to the wrong specification of when updates may happen, or how to transform the state. Roughly speaking, these can be broken down as timing errors and state transformation errors.

As an example timing error, suppose that a multi-threaded program attempts an update while thread $T_1$ holds a lock and thread $T_2$ waits for it. This may happen because the programmer allowed an update to happen at such a wrong
point in the program [40]. As a result, $T_1$ will block waiting for all threads to be ready to perform the update while $T_2$ is blocked waiting for the lock that will never be released.

A state transformation error can arise when the code to transform existing state is wrong. In our example, 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 the programmer mistakenly forgets to copy over the entries from the old table to the new version, leaving the latter uninitialized.2 As such, the new version may fail to find an entry on a subsequent GET that should have been present.

Post-update failures are an extreme threat to system availability, but a less extreme threat is the delay in service that occurs while an update is taking place. The delay is due to: (1) the time to quiesce the program threads, and (2) the time to transform an arbitrarily large program state. Part (1) can often be minimized [21], but part (2) may fundamentally take a while. For instance, in the example shown in Figure 1, the state transformation must iteratively update every existing entry in the table. If SIZE is large, this could take a long time.

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

Multi-Version Execution (MVE) allows several processes to execute in parallel over the same inputs to increase either reliability (a bug that affects only some of the versions is tolerated by the others which continue execution) or security (attacks 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 [27, 28, 30, 44, 49], 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 [28, 30, 49]. 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 OS. This is typically done using a ring buffer. The leader registers each system call and its result on the ring buffer. 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, and if so returning 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’s [28, 32, 36, 38]. The use of rewrite rules to tolerate divergences following an upgrade is a key element of MVEDSUA, which we describe next.

3.2 MVEDSUA: MVE-enhanced DSU

We illustrate how MVEDSUA enhances DSU with MVE in Figure 2. 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 §3.3. Eventually, the follower will drain the ring buffer, catching up with the leader.

Figure 2: MVEDSUA’s update stages.

---

2Not all DSU systems leave this task to the programmer, but Kitsune does.
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 demo- tion/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 can 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, MVEDSU A resumes the single-leader stage.

In sum, MVEDSU A 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 $t_2$), MVEDSU A avoids any pause in service availability.

**Handling in-update errors.** If the dynamic update fails prior to $t_2$, MVEDSU A 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 outdated leader stage. Bugs in the new version, or bugs in the update that are residual (e.g., owing to an incorrect state transformation), will manifest as a new-version crash or a divergence. In these cases, MVEDSU A terminates the follower and resumes single-leader mode with the old version until the bug is fixed and the update is retried.

**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. MVEDSU A recovers by promoting the new version to sole leader (jump to $t_6$). Such a promotion makes sense at any time after $t_3$. If the old version exhibits a bug between $t_3$ and $t_6$ that manifests as a divergence, then it (the follower) will be terminated.

---

Figure 3: Mapping that ensures old- and new-version states are related by the state transformation.

### 3.3 Maintaining a Consistent Semantic View

A key requirement for using MVEDSU A is for the programmer to provide a mapping from system call sequences issued by the leader to an equivalent sequence issued by the follower. During the outdated leader stage, the old version’s semantics are primary, and an old-version view of events must be given to the new version, to ensure they maintain an equivalent semantic view. During the updated leader stage, the new version’s semantics are primary, and the situation is reversed.

#### 3.3.1 Old-version Leader Mappings.

After the update is applied, MVEDSU A 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 3. The top line depicts the processing of client commands $cmd_1$, $cmd_2$, $cmd_3$, ... by the old-version leader. Each of these changes the leader’s state, depicted as a red shape, e.g., adding entries to the key-value store. The map downnarrow indicates the process by which system calls that correspond to these commands are mapped to system calls that correspond to commands $cmd_1'$, $cmd_2'$, $cmd_3'$, ... 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 downnarrow 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 both versions, it is likely that no syscall mapping is needed. This is the case for the `GET` and `PUT` commands, introduced in
read the Varan MVE’s DSL. It says that if a
either. Rule 1 in Figure 4 encodes this logic in the syntax of
rules to translate new commands that the leader does not un-
force the new version to adhere to behaviors
signal an actual error in the update or the new version; it is
a value. (cmd,typ) = parse($s);
$1 = "$cmd $key $val";
$1 += 7; }

(a) Updated follower $t_2$–$t_4$

// Rule 3
read(fd,s,n) {
  (cmd,typ) = parse($s$)
  return cmd == PUT & & typ == string; } as r
  => r(fd,s,n) {
    (cmd, _, _ ) = parse($s$);
    $1 = "$cmd $key $val";
    ($1) += 7; }

(b) Outdated follower $t_5$–$t_6$

Figure 4: Rewrite rules to map syscalls for PUT.

Figure 1. 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 under-
stand 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 = 1001 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, the programmer should
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 un-
derstand to commands that the follower does not understand either. Rule 1 in Figure 4 encodes this logic in the syntax of
the Varan MVE’s DSL. It says that if a read sys-
tem call receives a PUT command with a type component,
i.e., of the form PUT-type, then it should issue bad-cmd to
the new-version follower. The follower does not understand
this command and will reject it, just as the old version will
do for the original command. We only show the rule for the
PUT command for simplicity, the rules for the other com-
mands can be written in a similar way.

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, we have no choice but to terminate the
follower unless the command produces clearly-wrong be-
havior in the leader, such as a hang or a crash.

3.3.2 New-version Leader Mappings.

During the updated leader stage the new version is the leader
so the situation depicted in Figure 3 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 backward compa-
rable client API, there is little work involved. However,
for new commands it may be difficult or impossible to pro-
vide a proper mapping. For our example update, GET and
PUT commands will work as usual. But if the client sub-
mits 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 4. 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 check 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.

We have used added/removed/updated commands in our
examples to explain the expected mappings needed to handle
semantic differences between versions. Prior work by Aj-
mani et al. [7] identifies general principles for maintaining
a consistent semantic view when multiple nodes of a state-
ful, distributed system are interacting with clients of differ-
ent versions during a rolling upgrade. For example, they rec-
ommend exposing only the intersection of behaviors across
multiple versions so as to ensure that each client session’s
semantics is consistent with its own version. They enforce
this behavioral restriction by translating calls between ver-
sions when possible, and causing problematic calls to fail,
otherwise, mirroring the approach we have described here.
4. Implementation

We have implemented MVEDSUA by extending the Kitsune [25] DSU system with support from the Varan [28] MVE system. The bulk of our implementation is located inside Varan, in 1202 lines of extra C code. Kitsune required minimal changes, with only 88 lines modified. We believe that basing MVEDSUA on other general-purpose DSU systems would be straightforward.

The changes to Kitsune support coordinating with MVEDSUA 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 §5.3. 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. Pleasantly, this is something that Kitsune already does after quiescing at update points, so MVEDSUA pigbacks on that support.

MVEDSUA enhances Varan with efficient support for single-leader mode, which spans the 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 IDs (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 [28].

5. 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.

5.1 Vsftpd

Vsftpd is an open-source FTP server and is a useful benchmark because several other DSU systems have used it for evaluation [23, 25, 31, 33]. We used 14 versions 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 versions and number of rules required

<table>
<thead>
<tr>
<th>Versions</th>
<th># rules</th>
<th>Versions</th>
<th># rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.0 → 1.1.1</td>
<td>0</td>
<td>2.0.0 → 2.0.1</td>
<td>0</td>
</tr>
<tr>
<td>1.1.1 → 1.1.2</td>
<td>2</td>
<td>2.0.1 → 2.0.2</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 → 1.1.3</td>
<td>0</td>
<td>2.0.2 → 2.0.3</td>
<td>1</td>
</tr>
<tr>
<td>1.1.3 → 1.2.0</td>
<td>2</td>
<td>2.0.3 → 2.0.4</td>
<td>1</td>
</tr>
<tr>
<td>1.2.0 → 1.2.1</td>
<td>0</td>
<td>2.0.4 → 2.0.5</td>
<td>1</td>
</tr>
<tr>
<td>1.2.1 → 1.2.2</td>
<td>0</td>
<td>2.0.5 → 2.0.6</td>
<td>0</td>
</tr>
<tr>
<td>1.2.2 → 2.0.0</td>
<td>3</td>
<td>Average</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Figure 5: Rewrite rule for Vsftpd to safely redirect unknown commands to newer version.

(1) \text{read(}_{-1}\text{), write(}_{-1}"500 Unknown command"}_{-1}\text{)}

(2) \text{=} \triangleright \text{read(}_{-1}"FOOBAR\langle n\rangle",8), write(}_{-1}"500 Unknown command"}_{-1}\text{)}

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. Following the methodology in §3.3.1, we used a rule to trigger an invalid command on the new version while the old version is the leader. Figure 5 shows a general form of this rule: when the old version reports 500 Unknown command, it rewrites the STOU command to another invalid command, guaranteed to result in the same behavior in the new version.

An interesting, happy coincidence happens when the updated leader issues the new command STOU. Normally, this would cause an irreparable divergence (§3.3.2) and terminate the outdated follower. However, Vsftpd does not keep any state about the file-system that would diverge. Therefore, the leader executes the STOU command, generates a new file \textit{f}, and later GET commands of \textit{f} execute successfully on both leader and follower. We can thus write a rule to tolerate the STOU divergence on the follower, and the two versions will remain in sync. No existing MVE system keeps a separate copy of the file-system per version for performance reasons [27, 28, 30, 44, 49]. Of course, if an MVE system provided that extra separation, or if Vsftpd kept any state about the file-system (e.g., a cache), this solution would not work and MVEDSUA would terminate the outdated follower.

5.2 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 [25] and related systems [27, 42], and thus allows for a direct performance comparison. We also evaluated Redis with a bug fix to handle uninitialized read errors (7 lines per version, the same lines in the same files),
and 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.

5.3 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 [25] and related systems [42]. 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’s Kitsune support 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 §4), 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. Memcached’s event handling loop is inside LibEvent. When an update is signaled Kitsune interrupts LibEvent, which causes all threads to exit LibEvent, reach an update point, and then terminate (as Kitsune would relaunch LibEvent, which causes all threads to exit LibEvent, reach an update point). This allows LibEvent to reach an update point frequently, without exiting; which works for establishing quiescence when updating originally, and for swapping 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.

6. Experimental Evaluation

Here we describe how we used the applications described in §5 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.

6.1 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.

We performed this evaluation on a machine equipped with two Intel Xeon 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. 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 2B 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 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 results are in Table 2. The first six rows of the table show MVEDSUA’s performance in its two key modes of operation, single leader mode (MVEDSUA-1) and outdated/updated leader mode (MVEDSUA-2), as well as in related configurations which highlight component costs. The last three rows of Table 2 show the performance of competing techniques, explained in §7. MVEDSUA-1 represents a program’s normal mode of operation, and its overhead is small compared to a Native binary: 3%–9%. Mode MVEDSUA-2 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
Table 2: Steady-state performance and overhead of Memcached, Redis, and Vsftpd.

<table>
<thead>
<tr>
<th>Version</th>
<th>Memcached</th>
<th>Overhead</th>
<th>Redis</th>
<th>Overhead</th>
<th>Vsftpd small</th>
<th>Overhead</th>
<th>Vsftpd large</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ops/sec</td>
<td>Ops/sec</td>
<td></td>
<td>Ops/sec</td>
<td></td>
<td>Ops/sec</td>
<td></td>
<td>Ops/sec</td>
</tr>
<tr>
<td></td>
<td>×1000</td>
<td>×1000</td>
<td></td>
<td>×1000</td>
<td></td>
<td>×1000</td>
<td></td>
<td>×1000</td>
</tr>
<tr>
<td>Native</td>
<td>249 ± 1.05</td>
<td>73 ± 0.46</td>
<td>—</td>
<td>2535 ± 0.27</td>
<td>—</td>
<td>118 ± 0.06</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Kitsune</td>
<td>242 ± 1.52</td>
<td>74 ± 0.31</td>
<td>-1%</td>
<td>2594 ± 3.14</td>
<td>3%</td>
<td>117 ± 0.13</td>
<td>2%</td>
<td>—</td>
</tr>
<tr>
<td>Varan-1</td>
<td>234 ± 0.78</td>
<td>68 ± 0.92</td>
<td>8%</td>
<td>2458 ± 4.26</td>
<td>8%</td>
<td>116 ± 0.11</td>
<td>3%</td>
<td>—</td>
</tr>
<tr>
<td>MVEDSUA-1</td>
<td>226 ± 1.60</td>
<td>69 ± 0.12</td>
<td>6%</td>
<td>2048 ± 2.71</td>
<td>24%</td>
<td>90 ± 0.18</td>
<td>25%</td>
<td>—</td>
</tr>
<tr>
<td>Varan-2</td>
<td>125 ± 2.13</td>
<td>41 ± 0.46</td>
<td>44%</td>
<td>2001 ± 4.30</td>
<td>25%</td>
<td>89 ± 0.19</td>
<td>25%</td>
<td>—</td>
</tr>
<tr>
<td>MVEDSUA-2</td>
<td>121 ± 5.22</td>
<td>43 ± 0.11</td>
<td>42%</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MUC [42]</td>
<td>—</td>
<td>23.2%–87.1%</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Mx [27]</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>3x–16x</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Imago [16]</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

MVEDSUA when performing an update on Redis 2.0.0 → 2.0.1 and Memcached 1.2.2 → 1.2.3. In our experiments, the update happens at 120 seconds (t₁ in Figure 2); the promotion/demotion (t₄) happens at 180 seconds; and single leader mode resumes (t₆) at 240 seconds. Figure 6 shows how many operations Memtier completed per second, on average. Times t₁ and t₆ are immediately visible, with MVEDSUA entering and exiting MVE. The performance levels match MVEDSUA-1 and MVEDSUA-2 in Table 2. For Redis, t₄ 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: 2¹⁰, 2²⁰, and 2²⁴ entries.

Figure 7 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-2¹⁰, 6,900ms for MVEDSUA-2²⁰, and 260ms for MVEDSUA-2²⁴. 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.

6.2 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 7fbb16bac introduces an error that crashes the server when invoking command HMGET on the wrong type of data. The error is present on all versions of Redis we tested. As an experiment, we ran version 2.0.0 without the revision that introduced the error, so that the update 2.0.0 → 2.0.1 introduces it. Following a

3 Vsftpd is essentially stateless, which means its update-time pause is already low, so we did not measure it.
dynamic update with Kitsune, the program crashes when a client sends a bad \texttt{HMGET} command. But when using VEDSUA for the update, sending a bad \texttt{HMGET} command results in the follower failing, at which point execution reverts to single-leader mode. Clients proceed without incident.

\textbf{Error in the State Transformation.} A Kitsune-developed update had a latent bug: it would free memory still in use by LibEvent, which would cause a crash if the freed memory was 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 VEDSUA, however, this new-version crash is tolerated, and execution continues with the old-version leader without clients noticing.

\textbf{Timing Error.} Recall that we needed to change Memcached to reset LibEvent’s state in the leader after an aborted update (see §5.3). 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 VEDSUA. While we ultimately fixed the error by adding code to reset LibEvent’s state, the fact that diversions are aborted means we could have simply retried the update. In an experiment, we configured VEDSUA 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.

7. Related Work

MUC [42] 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 VEDSUA, 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 \texttt{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 huge overheads on syscall-heavy workloads. Second, MUC cannot handle failures during or after an update due to new-version or update-induced bugs, i.e. none of the faults listed in §6.2. 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, and has no good way to fix this issue.

POLUS [13] 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 1, POLUS could not back-transform store entries with non-string type.

TTST [18] proposes an approach for validating state transformations based on process-level updates. TTST first updates the old version (running in a process \textit{Old}) to the new version (running in a process \textit{New}) using forward state transformations, and then updates the new version to an old version (running in a process \textit{Reversed}) using backward state transformations. It then compares \textit{Old} and \textit{Reversed} in order to detect potential state transformation bugs, which would cancel the update. VEDSUA 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 VEDSUA 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 VEDSUA approach, by performing the forward and backward updates in the background.

Proteos [20] provides OS support to update a set of processes atomically. Similarly to VEDSUA, if the update fails, Proteos simply rolls it back, allowing the to-be-updated processes to resume execution in the old version. However, Proteos does not monitor the updated processes after the update is completed, so it cannot detect post-update errors such as those due to buggy patches. Furthermore, Proteos stops all processes while the update is taking place, so it cannot tolerate update-induced delays.

Mx [27] uses MVE to run two program versions in parallel and tolerate errors in one by using the other. However, Mx does not support DSU—it can run two versions with MVE from the beginning, but it cannot add a new version in mid-execution, resulting in fundamentally different requirements from VEDSUA’s. Mx executes versions in lock-step, synchronizing at each system call, so it incurs a significant overhead, e.g., 3-16x on a comparable scenario for the versions of Redis evaluated with VEDSUA. Finally, Mx does not tolerate system call divergences nor changes to data structures, which restricts its ability to deploy release-level versions. Mx was able to tolerate a simpler version of the new code error listed in §6.2, without syscall divergences; it fundamentally cannot tolerate the other two faults listed in §6.2.

Imago [16] 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, 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 [41, 45] or on-demand (lazy) state transformation [33, 37, 41], 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.

8. 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 DSU system that employs multi-version execution (MVE) to deliver a solution that both masks update pauses and tolerates a variety of failed updates. We implemented MVEDSUA by combining Kitsune, a modern DSU system, with Varan, a high-performance MVE system, and evaluated it on several high-performance servers—Redis, Memcached and Vsftpd. We found that MVEDSUA imposes low overhead in steady state, masks the update-time pauses, and tolerates real and realistic update errors.

References


