

Kitsune: Efficient, General-purpose Dynamic Software Updating for C

Christopher M. Hayden Edward K. Smith Michail Denchev
Michael Hicks Jeffrey S. Foster

University of Maryland, College Park, USA
{hayden,tedks,mdenchev,mwh,jfoster}@cs.umd.edu

Abstract

Dynamic software updating (DSU) systems allow programs to be updated while running, thereby permitting developers to add features and fix bugs without downtime. This paper introduces Kitsune, a new DSU system for C whose design has three notable features. First, Kitsune’s updating mechanism updates the whole program, not individual functions. This mechanism is more flexible than most prior approaches and places no restrictions on data representations or allowed compiler optimizations. Second, Kitsune makes the important aspects of updating explicit in the program text, making the program’s semantics easy to understand while minimizing programmer effort. Finally, the programmer can write simple specifications to direct Kitsune to generate code that traverses and transforms old-version state for use by new code; such state transformation is often necessary, and is significantly more difficult in prior DSU systems. We have used Kitsune to update five popular, open-source, single- and multi-threaded programs, and find that few program changes are required to use Kitsune, and that it incurs essentially no performance overhead.

Categories and Subject Descriptors C.4 [Performance of Systems]: Reliability, availability, and serviceability

General Terms Design, Languages

Keywords dynamic software updating

1. Introduction

Running software systems without incurring downtime is very important in today’s 24/7 world. *Dynamic software updating* (DSU) services can update programs with new code—to fix bugs or add features—without shutting them down. The research community has shown that general-

purpose DSU is feasible: systems that support dynamic upgrades to running C, C++, and Java programs have been applied to dozens of realistic applications, tracking changes according to those applications’ release histories [1, 5, 9, 12–14, 16, 17, 19]. Concurrently, industry has begun to package DSU support into commercial products [2, 20].

The strength of DSU is its ability to preserve program state during an update. For example, servers for databases, media, FTP, SSH, and routing can maintain client connections for unbounded time periods. DSU can allow those active connections to immediately benefit from important program updates (e.g., security fixes), whereas traditional updating strategies like rolling upgrades cannot. Servers may also maintain significant in-memory state; examples include memcached (a caching server) and redis (a key-value server). DSU techniques can maintain this in-memory state across the update, whereas traditional upgrade techniques will lose it (memcached) or must rely on an expensive disk reload that degrades performance (redis).

We are interested in supporting general-purpose DSU for single- and multi-threaded C applications. While progress made by existing DSU systems is promising, a truly practical system must be in harmony with the main reasons developers use C: control over low-level data representations; explicit resource management; legacy code; and, perhaps above all, performance. In this paper we present Kitsune, a new DSU system for C that is the first to satisfy these motivations while supporting general-purpose dynamic updates in a programmer-friendly manner. (We compare in detail against related systems in Section 5.)

Kitsune operates in harmony with C thanks to three key design and implementation choices. First, Kitsune uses entirely standard compilation. After a translation pass to add some boilerplate calls to the Kitsune runtime, a Kitsune program is compiled and linked to form a shared object file (via a simple Makefile change). When a Kitsune program is launched, the runtime starts a driver routine that loads the first version’s shared object file and transfers control to it. When a dynamic update becomes available (only at specific program points, as discussed shortly), the program longjumps back to the driver routine, which loads the new application version and calls the new version’s main function. Thus, ap-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA’12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10...\$10.00

plication code is updated all at once, and as a consequence, Kitsune places no restrictions on coding idioms or data representations; it allows the application’s internal structure to be changed arbitrarily from one version to another; and it does not inhibit any compiler optimizations.

Second, Kitsune gives the programmer explicit control over the updating process, which is reflected as three kinds of additions to the original program: (1) a handful of calls to `kitsune.update(...)`, placed at the start of one or more of the program’s long-running loops, to specify *update points* at which dynamic updates may take effect; (2) code to initiate *data migration*, where old-version data is assigned to new-version variables (possibly after *transformation* to reflect program changes); and (3) code to perform *control migration*, which redirects execution to the corresponding update point in the new version. In our experience, these code additions are small (see below) and fairly easy to write because of Kitsune’s simple semantics. (Section 2 explains Kitsune’s use in detail.)

Finally, Kitsune includes a novel tool called `xngen` that assists the programmer in writing code to migrate and transform old program state to be compatible with a new program version. The input to `xngen` is a series of programmer-provided *transformation specifications* (“transformers” for short), one per changed type or variable, that describe in intuitive notation how to translate data from the old to new format. The output of `xngen` is C code that performs state migration, executing transformation wherever it is needed. At a high level, the generated code operates analogously to a tracing garbage collector, traversing the heap starting at global variables and locals marked by the programmer. When the traversal reaches data requiring transformation, it allocates new memory cells and initializes them using the transformers, taking care to maintain the shape of the heap. The old version’s copies of any transformed data structures are freed once the update is complete. Kitsune’s approach is easy to use, relative to other DSU systems; it adds no overhead during the non-updating portion of execution; and it does not change data layout. (Section 3 describes `xngen`.)

We have implemented Kitsune and used it to update three single-threaded programs—`vsftpd`, `redis`, and `Tor`—and two multi-threaded programs—`memcached` and `icecast`. For each application, we considered from three months’ to three years’ worth of updates. We found that the number of code changes we needed to make for Kitsune was generally small, between 57 and 159 LoC total, across all versions of a program. The change count is basically stable, and not generally related to the application size, e.g., 134 LoC for 16 KLoC `icecast` vs. 159 LoC for 76 KLoC `Tor`. `xngen` was also very effective, allowing us to write transformers with similarly small specifications totaling between 27 and 200 lines; the size here depends on the number of data structure changes across the sequence of updates. We tested that all programs behaved correctly under our updates.

We measured Kitsune’s performance overhead and found it ranged from -2.2% to $+2.35\%$, which is in the noise on modern systems [15]. We also measured the overhead that Ginseng [16, 17] and UpStare [13], two other general-purpose DSU systems for C, imposed on some of the same programs and found it to be as high as 18.4% for the former and 41.6% for the latter. We also measured that the time required to perform an update using Kitsune at typically less than 40ms. One program, `icecast`, took ~ 1 s to update; this is due to internal timing constraints and does not adversely affect the application. (See Section 4 for full details.)

Kitsune’s design adopts the best ideas from existing systems while it eschews their shortcomings; we drew significant inspiration from UpStare and Ginseng. From the former, Kitsune adopts the notion of whole-program updates, rather than per-function updates, and from the latter it adopts the idea of updating only at explicitly specified update points, rather than at arbitrary positions. Whole-program updating eliminates the need to refactor programs to support certain updates (e.g., “loop extraction” [17] to enable updating long-running loops), while update points simplify the task of testing and otherwise reasoning about an update, since far fewer program states need to be considered.

On the other hand, Kitsune specifically rejects other elements of these systems’ designs to better balance competing concerns. For example, both UpStare and Ginseng require nontrivial compilers to enable updating—UpStare compiles the entire program specially to enable *stack reconstruction*, a mechanism that reduces the control migration problem to specifying a *stack mapping* at update-time, while Ginseng’s compiler inserts extra levels of indirection, adds “slop” space to **struct** definitions, inserts read/write barriers to support on-the-fly control and data migration, and performs a static analysis to ensure that these compilation changes will not break the program (e.g., due to tricky uses of typecasts).

Kitsune’s design requires the programmer to write slightly more code in the worst case compared to Ginseng and UpStare, but in general, confers several advantages, including: (1) significantly better performance, since control migration code is localized to program paths that rarely intersect with normal program execution, while Ginseng’s and UpStare’s compilation changes are pervasive; (2) simplified update understanding, since the programmer can just read the code without having to mentally apply a stack mapping and/or indirection model to it; (3) a simpler implementation, since no special compiler is needed; and (4) greater flexibility and scalability, since it does not require (conservative, slow) whole-program analysis that would prohibit certain programming idioms. In essence, Kitsune makes DSU a first-class *program feature* that is implemented and maintained by the programmer, a task that is made simpler and more manageable thanks to the careful design of the Kitsune run-time library and tool suite.

Section 5 provides a thorough, mechanism-level comparison to related DSU systems for C. Considered as a whole, we find Kitsune to be the most flexible, efficient, and easy to use (and deploy) DSU system for C developed to date.

2. Kitsune

A Kitsune application’s execution goes through three phases:

Normal execution. When started for the first time, and while no dynamic update is available, the application executes normally.

Update preparation. Once a dynamic update becomes available, the application thread(s) must reach a state in which the update can be safely applied. The programmer will insert calls to the function `kitsune_update` at program points at which an update is permitted to take effect; such calls are dubbed *update points* [12]. When an update is available, the `kitsune_update` function starts the update process. If the program is single-threaded, the new program is loaded and the next phase, update execution, begins. If the program is multithreaded, each thread blocks until all reach an update point, and then update execution begins.

Update execution. The threads running the old code have their stacks unwound, the entire new program is loaded, and its main function is called by the main thread. Since main also executes during normal startup, the programmer adds a few Kitsune API calls to direct it to behave differently during an update. This added code will do two things: (1) migrate and transform the old version’s data, and (2) direct control to a point in the new version that is equivalent to the point at which the update took place in the old version, identified by calls to `kitsune_update`. We call these two activities *data migration* and *control migration*, respectively. Once all threads have reached their update points, the update is complete, resources are freed, and normal execution resumes.

In what follows, we explain how we build programs to implement this semantics (Section 2.1), and how data and control migration is orchestrated by the programmer using the Kitsune API (Section 2.2). For simplicity, we start by assuming we are working with a single-threaded program, and conclude by describing how we handle multi-threaded programs (Section 2.3).

2.1 Implementing dynamic updating

The process of building a Kitsune application is illustrated in Figure 1. There are two inputs provided by the programmer: the main application’s `.c` source files (upper left) and an `xfgen .xf` specification file for transforming the running state during an update (not needed for the initial version). The source files are processed by the Kitsune compiler `kitc` to add some boilerplate calls derived from programmer annotations. Rather than compile and link the resulting `.c` files to a standalone executable, these files are compiled to be

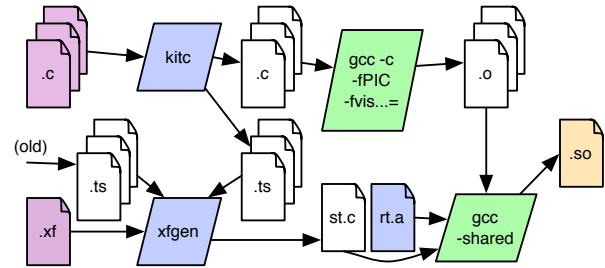


Figure 1. Kitsune build chain

position independent (using `gcc`’s `-fPIC` flag) and linked, along with the Kitsune runtime system `rt.a`, into a shared object library `app.so`. (For the best performance we also use `gcc`’s `-fvisibility=hidden` option to prevent application symbols from being exported, since exported symbols incur heavy overhead when called.) When building an updating version of the program, the `.xf` file is compiled by `xfgen` to C code and linked in as well. Processing the `.xf` requires `.ts type summary files` produced by `kitc` for the old and current versions (described in detail in Section 3.2).

The first version of a program is started by executing “`kitsune app.so args...`” where `args...` are the program’s usual command-line arguments. The `kitsune` executable is Kitsune’s application-independent *driver routine*, which dynamically loads the shared library and then performs some initialization. Among other things, the driver installs a signal handler for `SIGUSR2`,¹ which is later used to signal that an update is available. The driver also calls `setjmp`, and then transfers control to the (globally visible) `kitsune_init` function defined in `rt.a`. This function performs some setup and calls the application’s (non-exported) main function; at this point, *normal execution* begins. The `kitsune` driver is only 109 lines of C code and is the sole part of a program that cannot be dynamically updated.

When `SIGUSR2` is received, the handler sets a global flag; this starts the *update preparation* phase. The `kitsune_update` function will notice the flag has been set and call `longjmp` to return to the driver, which then dynamically loads the new program version’s shared object library. Since the `longjmp` call will reset the stack, the `kitsune_update` function copies any local variables marked for migration to the heap before jumping back to the driver. Thus, just after an update, the old version’s full state (e.g., its heap, open files and connections, process/parent ID, etc.) is still available. At this point, `kitsune_init` is invoked to start the new version, beginning the *update execution* phase. The programmer will have inserted calls to various Kitsune API functions to perform data

¹ The exact method for signaling that an update is available is left to the discretion of the programmer. However, Kitsune provides a sensible default of `SIGUSR2`, which works well in most cases. In programs we worked with, only Tor, which had a previously existing control framework, required a different mechanism.

```

1 /* config variables set by load_config() (code not shown) */
2 int config_foo, config_bar, config_size; /* automigrated */
3
4 typedef int data;
5 data *mapping; /* automigrated */
6
7 int op_count=0; /* automigrated */
8 struct dispatch_item
9 { char *key; dispatch_fn *fun; } dispatch_tab
10 __attribute__((kitsune_no_automigrate))
11 = { {"get", &handle_get }, {"set", &handle_set } };
12
13 void handle_set(int sock) {
14     key = recv_int(sock);
15     val = recv_int(sock);
16     mapping[key] = val;
17     send_response("%d>_ok", op_count);
18 }
19 void handle_get(int sock) {
20     key = recv_int(sock);
21     send_response("%d>_%d=%d", op_count, key, mapping[key]);
22 }
23 void client_loop(int sock) {
24     while (1) {
25         kitsune_update("client");
26         char *cmd = read_from_socket(sock);
27         if (!cmd) break;
28         dispatch_fn *cmd_handler = lookup(dispatch_tab, cmd);
29         op_count++;
30         cmd_handler(sock); }
31 }
32 int main() __attribute__((kitsune_note_locals)) {
33     int main_sock, client_sock;
34     kitsune_do_automigrate();
35     if (!kitsune_is_updating()) {
36         load_config();
37         mapping = malloc(config_size * sizeof(data)); }
38     if (!MIGRATE_LOCAL(main_sock))
39         main_sock = setup_connection();
40     if (kitsune_is_updating_from("client")) {
41         MIGRATE_LOCAL(client_sock);
42         client_loop(client_sock); }
43     while (1) {
44         kitsune_update("main");
45         client_sock = get_connection(main_sock);
46         client_loop(client_sock); }
47 }

```

Figure 2. Example; Kitsune additions highlighted

and control migration during this phase, which we illustrate in detail next. We consider `xfg` in Section 3.

2.2 Example

To use Kitsune, the programmer must slightly modify her application—to insert update points, and to add code to perform control and data migration. This subsection illustrates what these modifications look like, and describes the Kitsune API, using an example.

Consider the C program in Figure 2, which implements a simple key-value server. Clients connect to the server and send either `get i` to get the integer value associated with index i , or `set i n` to associate index i with value n . In the figure we have highlighted the extra code we need to perform data and control migration.

Let us ignore the highlighted code for the moment so that we can discuss the program’s core operation. Execution begins with `main()` on line 32. After defining some local variables, the function calls `load_config()` (code not shown) which initializes the three global configuration variables defined on line 2 and then allocates an empty mapping. Next, `main()` calls `setup_connection()` (code also not shown) to begin listening on `main_sock`. Finally, `main()` enters the main loop on lines 43–47. This loop repeatedly waits for a connection and then calls `client_loop()` to handle that connection. The `client_loop()` function repeatedly reads a command from the socket; finds the handler (a function pointer) for that command in `dispatch_tab` (created on lines 9–11); increments a global counter `op_count` that tracks the number of requests; and then dispatches to `handle_set` or `handle_get`. If the client disconnects, the function exits the loop on line 27 and returns. While this code is very simple, many server programs share this same general structure—a main loop that listens for connections; a client loop that dispatches different commands; and handler functions that implement those commands.

Now consider the highlighted code, which the developer has added to the program to implement Kitsune control and data migration. This code makes use of several primitive operations that Kitsune provides which we have summarized in Figure 3 and discuss here. We should emphasize that because this example is tiny, the amount of highlighted code is disproportionately large (see Section 4).

Migrating control. A dynamic update is initiated when the program calls `kitsune_update(name)`, where `name` identifies the update point, which can be queried when the new program version is launched. In Figure 2 we have added update points on lines 25 and 44, i.e., we have one update point at the start of each long-running loop. These are good choices for update points because the program is *quiescent*, i.e., in between events, when update-relevant state is not in the middle of being modified [8, 17].²

The kitsune driver will load the new version and call its main function. During update execution, the program will direct control toward the equivalent update point in the new version. To do this, it will branch on `kitsune_is_updating()`, which returns true if the program is being run as a dynamic update (or its variant `kitsune_is_updating_from(name)` for updates triggered at that named update point), to distinguish update execution from normal startup.

²Note that our definition of *quiescent* differs from (and is not comparable to) that of some prior work, which defines it to mean that all updated functions are *inactive*, i.e., not running.

In Figure 2, the conditional on line 35 prevents the configuration from being reloaded and mapping from being re-located when run as an update, since in this case the program will migrate that state from the old version instead (discussed below). If the update was initiated from the client loop, then on line 40 the program migrates `client_sock` from the previous version and then goes straight to that loop. Notice that when control returns from this call, the program will enter the beginning of the main loop, just as if it had returned from the call on line 46. Also notice we do not specifically test for an update from the "main" update point, as in that case the control flow of the program naturally falls through to that update point.

Migrating data. When a program using Kitsune starts update execution, critical data from the previous version of the program remains available in memory. The programmer is responsible for identifying what portion of that data should be migrated to the new version and specifying how that migration is to take place.

The first step is to identify the global and local variables that should be migrated. All global variables are migrated by default (that is, "automigrated"), and the programmer can identify any exceptions. For our example, migration occurs for the configuration variables on line 2 and for mapping on line 5. We use the `kitsune_no_automigrate` attribute on line 10 to prevent `dispatch_tab` from being automigrated, so that it is initialized normally—with pointers to new version functions—rather than overwritten with old version data. Local variables are *not* automigrated—the programmer must annotate a function with the `kitsune_note_locals` attribute (c.f. `main()`) to support migration of its local variables.

To facilitate data migration, `kitc` generates a per-file `do_registration()` function that registers the names and addresses of all global variables, including `statics`, and records for each one whether it is automigratable. The `do_registration()` function is marked as a constructor so it is called automatically by `dlopen`. Similarly, `kitc` introduces code in each of the functions annotated with `kitsune_note_locals` to register (on function entry) and deregister (on function exit) the names and addresses of local variables (in thread-local storage).

The second step is to indicate *when* data should be migrated after the new version starts. Calling `kitsune_do_automigrate()` (line 34) starts migration of global state, calling a *migration function* for each registered variable that is automigratable. These functions traverse data structures, transforming them wherever necessary, and are produced by `xfgen` automatically, when migratable data is unchanged between versions, or else according to programmer specifications. Each function follows a particular naming convention, and the runtime finds it in the new program version using `dlsym()`.

Within a function annotated with `kitsune_note_locals`, the user calls `MIGRATE_LOCAL(var)` to migrate (via the

appropriate migration function) the old version of `var` to the new version, e.g., as used on line 41 to migrate `client_sock`. `MIGRATE_LOCAL()` returns true if the program was started as a dynamic update; on line 38 we test this result to decide whether to initialize `main_sock`.

Our overall design for data migration reflects our experience that we typically need to migrate all, or nearly all, global variables, whereas we need only migrate a few local variables—only locals up to the relevant update point are needed, and of these, most contain transient state. We also assume that data that should be migrated is reachable from the application's local and global variables. In our experiments, this assumption was true except for `memcached`, in which pointers to some application data were stored only in a library. We solved this problem by caching such pointers in the main application; see Section 4.2.

Cleaning up after an update. After updating, Kitsune reclaims space taken up by the old program version. Since control and data migration are under programmer control in Kitsune, we need to specify the point at which the update is "complete." That point is when the new program version reaches the same update point at which the update occurred (c.f. the branch on line 42 of Figure 2, which then reaches the update point on line 25). Kitsune then unloads the code and stack data from the previous program version; to be safe, the programmer must ensure there are no stale pointers to these locations. For example, programmers must ensure any strings in the data segment that need to migrate are copied to the heap (which can be done in state transformers, or with `strdup` in the program text). Kitsune also frees any heap memory that `xfgen`-generated migrations have marked as freeable. Finally, control returns to the new version.

2.3 Multi-threading

Updating a multi-threaded program is more challenging since the programmer must migrate control and data for every thread. We could require the programmer to write this code manually, but we have observed that when the set of threads before and after the update is the same, a little additional support can make it easier to migrate those threads automatically.

To make a `pthread` program Kitsune-enabled, the programmer modifies all thread creation sites to use a wrapper for `pthread_create` called `kitsune_pthread_create`. A thread created with `kitsune_pthread_create(tid, f, arg)` has its thread id `tid`, thread function `f`, and `f`'s argument `arg` atomically added to a global list `kitsune_threads` of live threads. When a thread exits normally, it removes its entry from `kitsune_threads`.

Once an update becomes available, each non-main thread stops itself when it reaches an update point, recording the name of the update point in its `kitsune_threads` entry. When all threads have reached their update points, the main thread starts updating as described in Section 2.2, and continues un-

API call / attribute	Semantics during normal execution	Semantics during update execution
kitsune_update(label) kitsune_is_updating () kitsune_is_updating_from (label) kitsune_do_automigrate ()	Begins the update process when called, if a dynamic update is available Returns <i>false</i> Returns <i>false</i> Does nothing	Marks the completion of an update (so resources can be freed, etc.) Returns <i>true</i> Returns <i>true</i> if the update began from an update point with argument label Runs migration code to initialize the automigratable global variables
__attribute__ ((kitsune_no_automigrate))	Global variables <i>without</i> this attribute are noted so that they are migrated during the call to kitsune_do_automigrate ()	
__attribute__ ((kitsune_note_locals))	Local variables in a function with this attribute have their addresses registered when the function is called so they can be migrated should a new update begin before the function returns. If no update occurs, the addresses are deregistered when the function returns	
MIGRATE_LOCAL(localvar)	Returns <i>false</i>	Invokes migration code to initialize local variable localvar from its old version; returns <i>true</i>
MIGRATE_GLOBAL(globalvar)	Returns <i>false</i>	Invokes migration code to initialize global variable globalvar with kitsune_no_automigrate attribute; returns <i>true</i>
MIGRATE_LOCAL_STATIC(funcname, localvar) MIGRATE_GLOBAL_STATIC(globalvar)	As above, but for static global/local variables	

Figure 3. Kitsune primitives

til it finally reaches its own update point in the new version. Then the run-time system iterates through `kitsune_threads` and relaunches each thread, calling the new version of the recorded thread function with its recorded argument. If needed, the developer can provide a special transformation function to modify the set of threads or transform a thread’s entry function and argument. Each of those threads then executes, performing whatever control and data migration is needed. Each thread pauses when it reaches the update point where it was stopped. Once all threads have paused, the Kitsune runtime cleans up the old program version, releasing its code and data as usual, and resumes the main thread and all paused threads.

To update a multi-threaded program with Kitsune, that program should meet—or be modified to meet—several requirements. First, each long-running thread must periodically reach an update point. Typically this means a thread needs an update point in any long running loop and should avoid (uninterruptible) blocking I/O and similar operations. Second, threads should not hold resources, such as locks, at update points, since the thread could be killed and restarted at that point. This requirement is in keeping with the general criterion for choosing update points, which stipulates that little or no state should be in-flight. Third, the program should be insensitive to the order in which the threads are restarted in the new version. We expect this holds because the main thread will likely migrate any shared state, which would otherwise be the main source of contention between threads. Finally, recreating threads changes their thread IDs,

and so the program should not store those IDs in memory. (We could extend Kitsune to relax this requirement.)

All programs we considered in our experiments satisfy these requirements. In separate work, we performed a study to determine whether Kitsune’s approach would work for four additional programs, i.e., whether each thread would reach an update point in a timely manner [11]. In that study, we implemented additional runtime support to allow threads to wake from blocking calls (e.g., I/O and condition-variable-wait operations) and made small modifications to the programs (e.g., to break out of blocking calls in the `libevent` and `libpcap` libraries). We found that after these modifications, all of the threads in those programs were able to reach update points quickly (usually in less than 1ms).

Additionally, in our experiments we did not encounter concurrency-related problems during state migration. However, if such problems do occur in other programs, the developer can add synchronization to avoid interference.

In sum, all of the concurrency patterns we have observed so far have been compatible with Kitsune’s requirements.

3. xfggen

As mentioned briefly in Section 2.2, Kitsune’s runtime invokes migration functions for each automigrating variable, following a naming convention to locate the appropriate migration function. Kitsune includes a tool, `xfggen`, that produces migration functions automatically for variables and types that have not changed, and generates migration functions for those that have according to specifications the programmer expresses in a simple, domain-specific language.

<pre>INIT new_var: {action} INIT new_type: {action} old_var → new_var: {action} old_type → new_type: {action} old_var → new_var old_type → new_type</pre>	<pre>\$in, \$out – old/new type or var \$old/newtype(x) – x in old/new prog. \$old/newtype(t) – t in old/new prog. \$base – containing struct \$xform(old, new) – xformer function from old to new type/var</pre>	<pre>E_PTRARRAY(S) – size of ptr-to array E_ARRAY(S) – size of array E_OPAQUE – non-traversed pointer E_FORALL(@t) – polymorphism intro. E_VAR(@t) – refer to type var E_INST(typ) – instantiate poly. type</pre>
(a) transformers	(b) special variables	(c) type annotations

Figure 4. xfggen specification language and type annotations

We refer to such explicitly specified migration functions as *transformers*, since they are used to transform the data from an old representation to a new one. The design of xfggen is based on our experience applying DSU to C [8, 9, 12, 16, 17], and aims to make common kinds of state transformers easy to write while maintaining the flexibility to implement arbitrary transformations.

This section presents the xfggen specification language, giving a series of examples, and then describes how xfggen generates migration functions.

3.1 Transformer specifications

Figures 4(a) and (b) summarize xfggen’s specification language. Each transformer has one of the forms shown in part (a). The INIT transformers describe how to initialize new variables or values of new types, and the \rightarrow transformers describe how to transform variables or types that have changed and/or been renamed. Here $\{new,old\}_var$ is either a local or global variable name and $\{new,old\}_type$ is either a regular C type name or a **struct** field (we will see examples below). The transformer *action* consists of arbitrary C code that may reference the special xfggen variables shown in Figure 4(b). These variables refer to entities from the old or new program version. A \rightarrow transformation without an action identifies a variable/type renaming.

Example 1. Suppose we wrote a new version of the program in Figure 2 in which we removed the variable `op_count` and replaced it with two new variables `get_count` and `set_count` that record per-operation counts. These variables will need to be initialized during the update. We do not know exactly how many `get` and `set` operations have occurred, but we do have their sum in `op_count`, so we might over-approximate each with the sum, as follows:

```
INIT get_count: { $out = $oldsym(op_count); }
INIT set_count: { $out = $oldsym(op_count); }
```

Here we are initializing new variables, so we use an INIT transformer, and the action uses `$oldsym(op_count)` to refer to the old version’s value of `op_count` and `$out` to refer to the output of the transformer, i.e., `get_count` and `set_count` in the new version. Alternatively, we might wish to preserve the total count, and in lieu of precise information we might assume most calls are gets, and some are sets:

```
INIT get_count: { $out = (int) floor ($oldsym(op_count)*0.9); }
INIT set_count: { $out = (int) ceil ($oldsym(op_count)*0.1); }
```

In general, how a programmer writes a transformer depends on an application’s invariants and the desired properties of the updated application’s behavior [10].

Example 2. While transformers are often simple, xfggen is powerful enough to express more complicated changes. For example, suppose we change line 5 in Figure 2 so that, rather than an array, mapping is a linked list:

```
struct list {
    int key; data val; struct list *next;
} *mapping;
```

Then we can specify the following transformer:

```
1 mapping → mapping: {
2   int key;
3   $out = NULL;
4   for (key = 0; key < $oldsym(config_size); key++) {
5     if ($in[key] != 0) {
6       $newtype(struct list) *cur =
7         malloc(sizeof($newtype(struct list)));
8       cur→key = key;
9       cur→val = $in[key];
10      cur→next = $out;
11      $out = cur;
12 } } }
```

Here `mapping → mapping` indicates this is a transformer for the new version of `mapping` (the occurrence to the right of the arrow, referred to as `$out` within the action) from the old version of `mapping` (referred to as `$in`). The body of the transformer loops over the old `mapping` array (whose length is stored in old version’s `config_size`), allocating and initializing linked list cells appropriately. In the call to `malloc`, we use `$newtype(struct list)` to refer to the list type in the new program version.

Example 3. Finally, suppose the programmer wants to change type `data` from **int** to **long**, and at the same time extend `mapping` with field **int** `cid` to note which client established a particular mapping:

```
typedef long data;
struct list {
    int key; data val; int cid; struct list *next;
} *mapping;
```

The programmer can specify that `val` should simply be copied over and `cid` should be initialized to `-1`:

```
typedef data → typedef data: { $out = (long) $in; }
INIT struct list .cid { $out = -1; }
```

Because the type of mapping changed, `xfgen` will use these specifications to generate a function that traverses the mapping data structure, initializing the new version of mapping along the way. As we will see shortly, this is possible because there is a structural relationship between elements in the old list and elements in the new list, and because by default `xfgen`-created migrations stop traversal at `NULL`, the list terminator. (We could not use this approach for the previous array-to-list change because the data elements were not related in a simple structural manner.)

Other special variables. In the examples so far, we have seen uses of all but the last two special variables in Figure 4(b). The variable `$base` refers to the struct whose field is being updated. For example, in

```
INIT struct s.x: { $out = $base.y }
```

new field `x` of **struct** `s` is initialized to field `y` in the same **struct**.

Variable `$xform(old, new)` names the migration function between types `old` and `new`. This variable is useful when defining a transformer for a container datastructure, so that an action can recursively call the migration for each contained object. For example, suppose we merged Examples 2 and 3 into a single update that transformed mapping to a list and changed data's type to **long**. Then we could use the transformer from Example 2, changing line 9 to

```
XF_INVOKE($xform(data, data), &$in[key], &cur→val);
```

`$xform` looks up (or forces the creation of) the migration between its argument variables/types. This migration is returned as a closure that takes pointers to the old and new object versions and can be called using `XF_INVOKE`.

3.2 Migration generation

`xfgen` generates code to perform migration. At a high level, the generated code will traverse the heap, starting from the migratable global and local variables in the old version's stack and data segment, and assign the (possibly transformed) data to the corresponding variables in the new version. With `xfgen`, the programmer is able to focus on defining *what* transformation should be used for data representations that have changed, and is relieved of the tedium of *how* to find all of the old values, preserve the structure of the heap, and manage memory.

`xfgen` generates migration code based on the contents of the developer-provided `.xf` specification file and the Kitsune-maintained `.ts` type summary files for the old and new versions (see Figure 1). A type summary file contains all of the type definitions (e.g., **struct**, **typedef**) and global and local variable declarations from its corresponding `.c` source file, noting which are eligible for migration (according to the rules given in Section 2.2). To assist the programmer, `xfgen` can check that `.xf` files are complete: an `.xf` file is rejected if it fails to define a transformer that applies to migratable data that was added or whose type changed between versions.

`xfgen` uses type information to generate migration functions for migratable types and variables (or portions thereof) that have not changed—these functions will iterate over the variable/type in question, recursively invoking migration functions on the variable/type's subcomponents. Migratable data includes migrated local and global variables, their types, and types they transitively reference; e.g., **struct** `bar` is migratable if **struct** `foo` is migratable and contains a pointer to a **struct** `bar`. `xfgen` sometimes needs additional programmer-provided type information to work correctly; e.g., it may need to know the lengths of arrays.

In the remainder of this section, we describe how migration code is generated for variables and types, and how annotations are used to ensure the generated code is complete.

Migrating variables. For each migrated variable listed in the new version's `.ts` file, if that variable is named explicitly in an `old_var → new_var` specification, then `xfgen` generates C code from the specified action, substituting references appropriately. For example, `$in` and `$out` are replaced by values returned from `kitsune_lookup_old` and `_new`, respectively, which return a pointer to a symbol in the old or new program version, respectively, or `NULL` if no such symbol exists. Thus `xfgen` will produce the following C code from the overapproximating specifications in Example 1, above:

```
void _kitsune_migrate_get_count () {
  int *o_op_count = (int*) kitsune_lookup_old ("op_count");
  int *n_get_count = (int*) kitsune_lookup_new ("get_count");
  *n_get_count = *o_op_count;
}
void _kitsune_migrate_set_count () { /* as above */ }
```

For each remaining migrated variable `x`, `xfgen` will consult the `y→x` renaming rule if one exists to determine the source symbol `y`; otherwise it assumes `x`'s name is unchanged. In that case, as we noted previously, the lack of a symbol `x` in the old version code, or the lack of an explicit transformer if `x`'s type has changed, will cause `xfgen` to reject the `.xf` file. Assuming the old version symbol has type `old_type` and the new version has type `new_type`, `xfgen`'s generated function for `x` will simply call the migration function for `old_type → new_type`; if `old_type` and `new_type` have the same definition (and no explicit transformer has been specified) then `xfgen` will generate C code for this function. We describe this process next.

Migrating types. `xfgen` generates transformer code from `old_type → new_type` specifications in approximately the same manner as for variables. For specifications involving only a single **struct** field (as for the `cid` field in Example 3), `xfgen` will generate code for the rest of the fields in the manner described below, and then insert the hand-written code for the new or changed field.

A generated function for an unchanged type simply recursively invokes the migration functions for the immediate children of the type. For example, suppose we generated a migration for **struct** `list` in Example 3. Then, `xfgen` would

produce code that retains the old key value by copying it (generated migrations for primitive types are simple assignments); recursively invokes the user-provided transformer for data for the `val` field; inserts the user-provided code to assign to the `cid` field; and recursively invokes itself on the target of the next field, assuming it is not `NULL`.

Pointers must be handled carefully by the generated code. For non-`NULL` pointers, the generated code checks a global *migration map* to see if the pointer has been migrated before; if so it returns the old target. Doing so maintains the shape of the heap and avoids infinite loops when traversing cyclic datastructures. Otherwise it calls the appropriate migration for the pointer’s target. If the pointer is to a global or local variable, then the address of this variable is different in the new version. As such, the code will look up the corresponding address and redirect to it. If the pointer target’s type has truly changed (and so must have an explicit transformer), the generated code mallocs space to store the result. In that case, the old-version pointer is added to a list of addresses to be reclaimed once the update is complete. This is what happens for `next` in Example 3—since the **struct** increased in size, new memory is allocated for it and all fields must be copied. As an optimization, since the generated code for an unchanged **struct** reuses the old memory, migrations ignore non-pointer fields, retaining the old values. Once the source and target addresses (which may be the same) have been determined, they are added to the migration map. Note that using `malloc/free` for memory management during migration precludes programs that use custom allocators. We hope to address this limitation in future work.

`xfgen`-generated code may uselessly traverse portions of the heap that do not contain changed data. If the programmer knows that a particular data structure contains only pointers into the heap (and not to global or local variables), and that no pointed-to objects require transformation, she can create transformers that truncate the traversal. For example, if field `f` of **struct** `foo` (transitively) points only to unchanged heap data, the developer could write

```
struct foo.f → struct foo.f: { $out = $in; }
```

to shallow-copy the field. We did this for `redis-mod` (Figure 5, discussed in Section 4.3).

The migrations generated by `xfgen` assume there are no pointers into the middle of migratable objects. To help check this assumption, we provide an execution mode in which the created migrations use an interval tree to record the start and end of each object they encounter. A migration reports an error if it is ever asked to migrate an object that overlaps with, but does not exactly match the bounds of, a previously migrated object. Supporting pointers to the interior of objects is future work.

Type annotations. `xfgen` sometimes needs type information beyond what is normally available in C. For example, without further guidance, `xfgen` would generate an incorrect

migration for mapping in Example 1. It would assume that mapping points to a single data element, rather than an array of elements. In Kitsune, this extra information is provided by the programmer as annotations, shown in Figure 4(c). `kitc` recognizes these annotations and adds the information supplied by them to the `.ts` files.

The annotations, inspired by `Deputy` [6], are straightforward. `E_PTRARRAY(S)` provides a size `S` for a pointed-to array. For example, we would change line 5 of Figure 2 to

```
data * E_PTRARRAY(config_size) mapping;
```

By default, `xfgen` assumes that `t*` values for all types `t` are annotated with `E_PTRARRAY(1)`; explicit annotations override this default. Annotation `E_ARRAY(S)` provides a size `S` for array fields at the end of a **struct** (such fields can be left unsized in C). For both of these annotations, `S` can be an integer constant, a global variable, or a co-located **struct** field. `E_OPAQUE` annotates pointers that should be copied as values, rather than recursed inside during traversals (so we could use this annotation on the `foo.f` field to truncate traversal in the above example, rather than define a manual transformation).

Finally, `xfgen` includes annotations to handle some idiomatic uses of `void*` to encode parametric polymorphism (a.k.a. generics). For example, the following definition introduces a **struct** `list` type that is parameterized by type variable `@t`, which is the type of its contents:

```
struct list {
    void E_VAR(@t) *val;
    struct list E_INST(@t) *next;
} E_FORALL(@t);
```

`E_FORALL(@t)` introduces polymorphism, `E_VAR(@t)` refers to type variable `@t`, and `E_INST(@t)` instantiates a polymorphic type with type `@t`. For comparison, this example is equivalent to the following Java generic linked list:

```
class List<T> { // like E_FORALL(@T)
    T val; // like E_VAR(@T)
    List<T> next; // like E_INST(@T)
}
```

With generics, we can write **struct** `list E_INST(int) *x` to declare that `x` is a list of `ints`. Generated migration code will invoke the migration function that is appropriate for the list elements’ instantiated type.

4. Experiments

To evaluate Kitsune, we used it to develop dynamic updates for five widely deployed server programs.

To quantify the programming effort of using Kitsune, we tallied the number and kinds of changes we made to the programs, and the number and variety of `xfgen` specifications we wrote for state transformation. Overall, we made few code changes—between 57 and 159 LoC—to support updating, with most changes only to the initial version. Likewise, `xfgen` specifications were generally small, averaging

Program	# Vers	LoC	Upd	Ctrl	Data	E.*	Oth	Σ	$v \rightarrow v$	$t \rightarrow t$	Σ	xf LoC
<i>vsftpd</i>	14 (1.1.0–2.0.6)	12,202	6	26	17+8	6+14	28+8	83+30	9	21	30	101
<i>redis</i>	5 (2.0.0–2.0.4)	13,387	1	2	3	43	8	57	0	4	4	37
<i>Tor</i>	13 (0.2.1.18–0.2.1.30)	76,090	1	39	37+6	19	57	153+6	16	15	31	189
<i>memcached*</i>	3 (1.2.2–1.2.4)	4,181	4	9	13	20	66	112	12	10	22	27
<i>icecast*</i>	5 (2.2.0–2.3.1)	15,759	11+1	22+3	14+9	32+3	39	118+16	25	50	75	200

*Multi-threaded

Table 1. Kitsune benchmark programs, and modifications to support updating

3–4 lines per changed variable or type. These numbers are comparable to prior work.

To assess the performance overhead of using Kitsune, we measured the slowdown on normal execution for the Kitsune versions of the servers compared to the originals. We also measured the time taken to perform an update, from signaling to completion. We find that there is essentially no overhead on normal execution, a result uniformly better than prior work. We found that the time required to apply an update ranges from 2ms up to 1s, depending on the program; in all cases, the times seem acceptable for typical use.

4.1 Benchmarks

We chose a suite of benchmark programs that maintain in-process state that would be beneficial to preserve during an update. *Vsftpd* is a popular open-source FTP server. *Redis* is a key-value database used by several high-traffic services, including *guardian.co.uk* and *craigslist.org*. *Tor* is a popular onion-router that provides anonymous Internet access. *Memcached* is a widely used, high-performance data caching system employed by sites such as Twitter and Wikipedia. *Icecast* is a popular music streaming server. All of these programs maintain persistent network connections that an offline update would interrupt. Redis and memcached also maintain potentially large volumes of in-memory data that would either be lost (memcached) or expensive to restore (redis) following an update. Vsftpd also serves as a useful benchmark because several other DSU systems have used it for evaluation [5, 9, 13, 17].

The left portion Table 1 lists for each program the length of the version streak we looked at (for n versions, there are $n-1$ updates), which versions we considered, and the number of source lines of the last version as computed by *sloccount*. We consider at least three months of releases per program; for Tor we cover two years and for vsftpd we cover three. We tested that all programs behaved correctly before, during, and after updates were applied.

4.2 Programmer effort

Here we describe the manual effort that was required to prepare these programs for updating with Kitsune, and to craft updates corresponding to releases of these programs.

The center portion of Table 1 summarizes the Kitsune-related changes we made to these programs, tabulating the number of update points added; the number of lines

of code needed for control migration and data migration;³ the number of type annotations for xfggen; the number of lines changed for other reasons; and their sum. Each column shows the number of changes in the first version, followed by $+n$ where n is the sum of changes in all subsequent versions; if this is omitted, no further changes were needed.

One striking trend in the table is that most required changes occurred in the first version. Control migration and update points were particularly stable, essentially because the top-level control structure of the programs rarely changed. Data migration code and annotations were occasionally added along with new data structures. Another interesting trend is that the magnitude of the changes required is not directly proportional to either the code size or number of versions considered, e.g., 134 LoC for 16 KLoC icecast vs. 159 LoC for 76 KLoC Tor. On reflection, this trend makes sense. Changes to support control migration depend on the number and location of update points, and data annotations depend on the type and number of data structures; none of these characteristics scales directly with code size. Together, these numbers show that with Kitsune, DSU can be viewed as a stable program feature that is added to the program and maintained (with minimal comparative effort) as the program evolves.

The rightmost columns of the table list the xfggen specifications we wrote for each program’s updates. In particular, we list the number of variable transformers ($v \rightarrow v$) and type transformers ($t \rightarrow t$), across all versions, and their sum. We also list the total number of lines of transformer code we wrote, across all versions. We can see that, on average, 3–4 lines of xfggen code were needed for each transformer.

Overall, we found both the control and data migration code relatively easy to write. Following the structure of the example given in Section 2.2, we added conditionals to avoid re-initializing data we wished to preserve across the update, and to direct control back to the correct update points. State transformation code was also relatively easy to write. Most state required either no or very simple transformation along the lines of Example 1 in Section 3.1. Perhaps the most tricky part was adding type annotations to data structures.

³Specifically, control migration changes comprise calls to `kitsune_is Updating` and `kitsune_is Updating From`. Data migration changes include calls to `kitsune Do Automigrate`, `MIGRATE GLOBAL`, and `MIGRATE LOCAL`, and uses of the `kitsune No Automigrate` attribute.

While in many cases these annotations were obvious (specifying generic types or bounded arrays) or prompted by xfggen, a missing annotation was sometimes hard to debug. For example, failing to annotate a pointer as an array would result in the generated code not migrating all of the state, ultimately leading to a later crash. Fortunately, since Kitsune uses normal compilation and linking, we could use gdb to debug these problems directly.

Now we consider the particulars of each program, and when possible, compare the magnitude of these changes against those required by prior systems. In general, using Kitsune seems to require more program changes than prior systems, but the total sum of changes is still small.

Vsftpd. Many of the changes we made to vsftpd were typical across our benchmarks: we added type information for generics and inserted control flow changes to avoid overwriting OS state when updated. We added one update point for each of the five long-running loops in the program, which comprise the connection acceptance loop, two loops to implement privilege-isolated logins, the main FTP command processing loop, and a loop left running in a privileged parent process that implements commands such as chown.

The most interesting change we made to vsftpd was to handle I/O. Vsftpd replaces calls to `recv` with calls to a wrapper that restarts the actual read if it is interrupted, e.g., by the receipt of a signal. We inserted one update point in the wrapper so that interruption can initiate an update. To simplify the control-flow changes needed, rather than give the update point its own name, we reused the name of the update point in the loop that initiated the wrapped call; this is safe because this loop will reinitiate the call when the update completes.

Other DSU systems. Neamtiu et al. [17] applied Ginseng, another DSU system, to vsftpd. They updated a subset of the version streak we did (finishing at version 2.0.3). Even though their changes support just one update point (versus our six, which permit updating in many more situations), the effort was comparable: They report 50 LoC changed and 162 lines for state transformation, compared to 113 LoC changed and 101 lines of state transformation for Kitsune.

Makris and Bazzi [13] also updated vsftpd using UpStare for a shorter streak. They say that “some manual initialization of new variables and struct fields” was required, along with “11 user-defined continuation mappings,” but provide no detail as to their overall size. The UpStare distribution includes 14 mapping files, containing 4,644 lines of C code between them. Much of this code appears auto-generated. The manually written portion is chiefly used to transform variables and types.

Redis. Redis required few modifications to support updating. We placed a single update point in its main event loop and added one check to avoid some reinitialization. The vast majority of redis’s state is stored in a single global variable, `server`, so few variables needed migration. Redis makes ex-

tensive use of linked lists and hash tables, and we used xfggen’s generics annotations to model their types precisely. The version streak we considered included only code modifications, but we still needed xfggen to migrate data structures that reference global variables (whose addresses change with each updated version). For example, we wrote a 19-line specification to direct the traversal through a `void*` field by consulting an integer key to determine the field’s type; while long, this code was straightforward. Finally, redis uses a custom allocator, which xfggen does not support, so we modified the redis header files with preprocessor directives to redirect custom allocator calls to `malloc` and `free`.

We are unaware of prior work applying DSU to redis.

Tor. Tor is the largest of our benchmark programs, at ~76 KLoC. Adding DSU support required one update point in Tor’s main loop. The larger number of control flow changes in comparison to other programs is a result of Tor’s modular design. Twelve of the 33 modules’ `module_init` functions required simple two-line changes to prevent reinitialization of updated state. The remaining changes were made along the path from main to Tor’s main loop, similarly to our other benchmarks. Most “data” changes were `kitsune_no_automigrate` attributes, directing Kitsune to skip over various constant strings used in parsing Tor’s configuration file. We automigrated most global variables at the start of update execution, but manually migrated Tor’s `network_consensus` data structure (via a call to `MIGRATE_GLOBAL`) which, for an update that added a new field, required other global state to be consistent first. The rest of our changes (counted above as “other”) to Tor were primarily to add support to initiate an update via Tor’s control interface, rather than a command-line signal, for easier testing.

We wrote eight xfggen rules, corresponding to 16 variables and 15 types. All the transformers for Tor data structures were straightforward, since type representation changes were rare in the streak we considered. Investigating further, we found that Tor’s data representations tended to remain stable because most Tor data represents protocol messages, and these rarely change so as to preserve backward compatibility. The bulk of the remaining xfggen rules update function pointers for event handling stored in `libevent` and/or `OpenSSL` datastructures.

We are unaware of prior work applying DSU to Tor.

Memcached. Memcached is a multi-threaded server implemented using `libevent`. Unlike Tor, memcached’s main loop is in `libevent`’s `event_base_loop` function. To work with Kitsune, we had to change main to install a `libevent` callback on `SIGUSR2` to handle update notifications. When the main thread is notified of an update, it uses pipes, via `libevent` itself, to notify each of the child threads of the update; each thread then terminates in response. The main thread then performs the update and all threads work back to calls to `event_base_loop` in the new code. Interestingly, we needed to “sandwich” such calls with like-named update points:

```
kitsune_update("upd"); /* complete any active update */
event_base_loop(libevent_base, 0); /* pass control to libevent */
kitsune_update("upd"); /* a new update upon return */
```

When a thread is signaled, the `event_base_loop` call returns and initiates the update. When the program restarts, the update will complete when it reaches the update point (of the same name) just prior to the `event_base_loop` call in the new code.

There are two additional challenges in updating due to `libevent`: First, as with `Tor`, we needed to reinstall new function pointers in `libevent` after an update. Second, `memcached` installs the data associated with active connections in `libevent`, but does not retain its own pointers to that data. To enable updates to transform that data, we added code to maintain, in `memcached` itself, a list of active connections, which we can then use for state transformation.

Other DSU systems. Neamtiu and Hicks [16] updated `memcached` using `Ginseng`. They needed 26 lines of program changes and 12 lines for state transformation. `Kitsune` required more changes in part because we did not change `libevent` itself, which in Neamtiu and Hicks’ setup was merged into the main program (and thus was updatable). Their changes also created a problem with reaching update points suitably often due to intervening blocking calls; placing the update point outside `libevent` avoided this issue.

Icccast. `Icccast` is another multi-threaded program, with separate threads for connection acceptance, connection handling, file serving, receiving a stream from another server, sending statistics, and more. To modify `icccast` to support DSU with `Kitsune`, we located each of the thread creation points and then inspected the thread entry function and related code to make two types of changes. First, we identified the thread’s long-running, event-handling loop to which we added a call to `kitsune_update`. Second, we added the necessary annotations to migrate local variables or skip initialization during thread startup. The most complex `icccast` patch added a new thread to handle authentication (which required us to add an update point to the new authentication thread’s code) and reduced the number of connection threads. To implement this patch, we wrote transformation code that uses a C API provided by the `Kitsune` runtime that exposes the set of active threads at the time the update was taken (including the entry function, initialization argument, and update point taken) and allows the developer to add and remove threads or modify the properties of existing threads.

Other DSU systems. Neamtiu and Hicks [16] also considered updates to the same streak of `icccast` versions. They changed 154 LoC and wrote 80 lines of state transformation code. For `Kitsune` we changed 134 lines of the main program, and wrote 200 lines of `xfgen` specifications. A large proportion of these specifications were simple rules to initialize added variables or fields. However, they also included more complex rules for adding and removing a thread, as

Program	Orig (sqr)	Kitsune	Ginseng	UpStare
<i>64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7</i>				
vsftpd 2.0.6*	6.55s (0.04s)	+0.75%	–	–
memchd 1.2.4	59.30s (3.25s)	+0.51%	–	–
redis 2.0.4	46.83s (0.40s)	-0.31%	–	–
icccast 2.3.1	10.11s (2.27s)	-2.18%	–	–
<i>32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10</i>				
vsftpd 2.0.3*	5.96s (0.01s)	+2.35%	+11.3%	+41.6%
vsftpd 2.0.3 [†]	14.03s (0.02s)	+0.29%	+1.47%	+6.64%
memchd 1.2.4	101.40s (0.35s)	-0.49%	+18.4%	–
redis 2.0.4	43.88s (0.16s)	-1.21%	–	–
icccast 2.3.1	35.71s (0.68s)	+1.18%	-0.28%	–

*CD+LS benchmark, [†]file download benchmark

Table 2. Normal execution performance overhead

mentioned above. `Ginseng`’s patches do not attempt to remove this extra connection thread.

4.3 Performance

Normal execution overhead. We measured the overhead `Kitsune` adds to normal execution for all programs except `Tor`, discussed separately below. For comparison, we also measured the overhead of `Ginseng` and `UpStare` on programs they had previously benchmarked: `vsftpd`, `memcached`, and `icccast` for `Ginseng`, and `vsftpd` for `UpStare`.

We used the following workloads: For `memcached`, we ran `memslap` (2.5M operations using `memslap`’s default workload). For `redis`, we used `redis-benchmark` (1M GET and 1M SET operations), and for a fair comparison, we modified the non-updating version of `redis` to use the standard memory allocation functions, as we had done to support `xfgen`. (Note that switching to standard `malloc` slightly improves `redis`’ performance, since the custom allocator performed extra bookkeeping for tracking memory usage.) For `vsftpd`, we performed two separate tests. First, we measured the time to perform the following interaction 2K times: connect to the server, change directories, and retrieve a directory listing. Second, for the 32-bit platform, we measured the time to establish a connection and download a 32-byte binary file 200 times. This test closely resembles tests reported in the `UpStare` and `Ginseng` papers [13, 17]. For `icccast`, we used a benchmark originally developed for `Ginseng` [16] that measures the time taken for 16 simultaneous clients to download 7 music files, each roughly 2MB in size. For all programs, we ran the client and server on the same machine, to factor out network latency.

Table 2 reports the results. We ran each benchmark 21 times and report the median time for the unmodified programs along with the semi-interquartile range (SIQR), and the slowdowns for `Kitsune`, `Ginseng`, and `UpStare` (the median time for each compared to the median original time). The top of the table gives results on a 24 core, 64-bit machine, and the bottom gives results on a 2 core, 32-bit machine. `Ginseng` only works in 32-bit mode and `UpStare` is

Program	Med. (siqr)	Min	Max
64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7			
vsftpd →2.0.6	2.99ms (0.04ms)	2.62	3.09
memcached →1.2.4	2.50ms (0.05ms)	2.27	2.68
redis →2.0.4	39.70ms (0.98ms)	36.14	82.66
icecast →2.3.1	990.89ms (0.95ms)	451.73	992.71
icecast-nsp →2.3.1	187.89ms (1.77ms)	87.14	191.32
tor →0.2.1.30	11.81ms (0.12ms)	11.65	13.83
32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10			
vsftpd →2.0.3	2.62ms (0.03ms)	2.52	2.71
memcached →1.2.4	2.44ms (0.08ms)	2.27	3.12
redis →2.0.4	38.83ms (0.64ms)	37.69	41.80
icecast →2.3.1	885.39ms (7.47ms)	859.00	908.87
tor →0.2.1.30	10.43ms (0.46ms)	10.08	12.98

Table 3. Kitsune update times

only distributed in binary packages, which are only available for 32-bit systems.

From this data, we can see that Kitsune adds essentially no overhead to normal execution: the performance differences range from -2.18% to 2.35%, which is well within the noise on modern systems [15]. In contrast, the overhead for Ginseng and UpStare is more significant: for Ginseng it is 11.3% and 18.4% for memcached and the first vsftpd benchmark, respectively, and for UpStare it is 41.6% for the first vsftpd benchmark. For vsftpd, this overhead is higher than previously reported [13, 17]: Ginseng’s overhead was reported at 3% and UpStare’s at 16%. These results are close to those of our second vsftpd benchmark— 1.47% overhead for vsftpd and 6.64% for UpStare—giving us confidence that we are making a fair comparison. We conjecture that the higher overhead in the first benchmark is due to it spending a higher proportion of time executing application code than the second, and thus is more impacted by UpStare’s and Ginseng’s special compilation. For comparison, UpStare’s previously reported overhead on PostgreSQL (with data stored in memory to spend less time in OS code) was over 40%.

Tor. While we did not measure the overhead of Kitsune on Tor directly, we did test it by running a Tor relay in the wild. We dynamically updated this relay from version 0.2.1.18 to version 0.2.1.30 (13 versions) as it was carrying traffic for Tor clients. We initiated several dynamic updates during periods of load, when as many as four thousand connections carrying up to 11Mb/s of traffic (up and down) were live. No client connections were disrupted (which would have been indicated by broken or renegotiated TLS sessions). Over the course of this experiment, our relay carried 7TB of traffic.

Time required for an update. We also measured the time it takes to deploy an update with Kitsune, i.e., the total elapsed time for an update’s preparation and execution, starting from when the update is signaled to when it has completed. Table 3 summarizes the results for the last update in each streak, giving the median+SIQR, minimum, and maximum update times. For each program, we picked a suitable work-

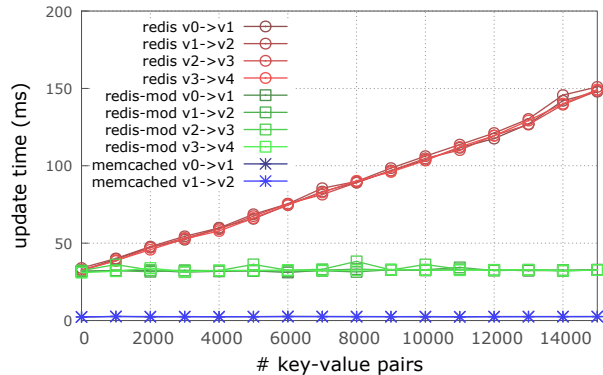


Figure 5. State size vs. update time (color plot)

load during which we did the update. For vsftpd, we updated after an FTP client had connected to and interacted with the server; for redis and memcached, we inserted 1,000 and 15,000 key-value pairs, respectively, prior to update; and for icecast, we established one connection to a music source and 10 clients receiving that stream prior to updating. For Tor, we fully bootstrapped as a client, establishing multiple circuits through the network and communicating with directory servers, and then applied the update.

For all programs except icecast, the update times are quite small. For icecast, most of the nearly 1 second delay occurs while the Kitsune runtime waits for each thread to reach an update point. This time was lengthened by one-second sleeps sprinkled throughout several of these threads. The line in the table labeled *icecast-nsp* measures the update time when these sleeps are removed, and shows the resulting time is much shorter. Because the sleeps are there, we conjecture icecast can tolerate the pause for updates; we did not observe a noticeable stop in the streamed music during the update. In recent work [11], we have developed techniques to support faster update times, showing significant improvements for icecast in particular. We plan to port these ideas to Kitsune in the near future.

Recall from Section 3.2 that xfgn-generated migrations may traverse significant portions of the heap, and thus for some updates the update time may vary with the size of the program state. Among our programs, the most likely to exhibit this issue are redis and memcached, as they may accumulate significant amounts of state. Figure 5 graphs the update time for these two programs versus the number of key-value pairs stored. For redis, the update time grows linearly because we traverse each of the data items on the heap, since some contain pointers to global variables that must be redirected to the new version’s data segment. On the other hand, memcached’s update times remain relatively constant because it stores its data in arrays that we treat opaquely, removing the need to traverse each instance.

Examining redis more closely, we observed that the pointers that force us to traverse the heap in fact point to a small, finite set of static locations. Thus, we created a modified (42 LoC changed) version of redis, labeled redis-mod, that stores integer indices into a table in place of those pointers. This obviates the need for a full heap traversal for all the updates in our streak, allowing updates times to remain constant for the tested heap sizes. Programs that use Kitsune may benefit from a similar transformation if they maintain a large amount of state containing static pointers.

5. Related Work

In this section we consider recent work that supports DSU for programs written in C and C++; these languages impose stringent constraints on a DSU system’s design. Table 4 characterizes the mechanisms used to implement Kitsune and other recent C/C++ DSU systems. Ekiden [9], Ginseng [17], OPUS [1], POLUS [5], and UpStare [13] target applications, while Ksplice [2], K42 [3], LUCOS [4], and DynaMOS [14] support (or are) OS kernels. (LUCOS is essentially a version of POLUS that uses VMMs to effect changes in operating systems; all comments we make about the latter apply to the former.) We discuss tradeoffs resulting from these mechanism choices, and argue that Kitsune provides the greatest flexibility and best performance with modest programmer effort. The footnotes in the table summarize the discussion below. (A direct comparison to two systems, Ginseng and UpStare, appears at the end of the introduction.)

Code updates. Most systems effect code updates at the granularity of individual functions (or objects). As noted in the first column, Ksplice, OPUS, DynaMOS, and POLUS insert, at run-time, a trampoline in the old function to jump to the function’s new version.⁴ As noted in the second column, Ginseng and K42 use indirection: Ginseng compiles direct function calls into calls via function pointers, while the K42 OS’s object handles are indirected via a hand-coded *object translation table* (OTT); updates take effect by redirecting indirection targets to the new versions.

There are several drawbacks to using these mechanisms. Trampolines require a writable code segment, which makes the application vulnerable to code injection attacks. Trampoline-based updating may break programs optimized using inlining, since it presumes to know where the start of a function is, so POLUS and OPUS both forbid inlining. (Ksplice is able to account for the compiler’s inlining decisions.) Using indirect calls adds overhead to normal execution and also inhibits inlining. Most onerously, neither trampolines nor indirections support updating functions that never (or rarely) exit, such as main, which changes relatively frequently [8], or functions that contain event-handling loops, such as the

⁴The first instruction of the function is replaced by a jump to a small piece of code, the *trampoline*, that executes the replaced instruction and then jumps to the function’s new version.

	Code upd			Data upd			Timing	
	tramp	ind	prog	repl	shdw	wrap	nonactv	upd pts
DynaMOS ⁴	×				×			
Ekiden			×	×				×
Ginseng ¹²³⁴⁵		×				×		×
K42 ²⁵		×		×			×	
Ksplice	×				×		×	
OPUS ²	×			-	-	-	×	
LUCOS/POLUS ²⁴⁵	×			×				
UpStare ²³			×	×				(×)
Kitsune			×	×				×

¹needs deep analysis

²inhibits optimizations

³pervasive instrumentation

⁴mixes old and new code

⁵relaxed thread sync.

Table 4. Comparing DSU systems for C/C++

scheduling loop in the OS. In the best case, programmers must refactor the program to place long-running loop bodies in separate functions (e.g., using “loop extraction” [17]).

The remaining three systems, UpStare, Ekiden, and Kitsune support more general changes by updating at the granularity of the whole program rather than individual functions. UpStare loads in code for the new program and then performs *stack reconstruction*: the running program automatically *unwinds* the current stack one function at a time back to main, and then *rewinds* the stack to a new-version program point specified by the programmer. In contrast, Kitsune relies on the programmer to migrate control to the equivalent new-version program point.

Kitsune’s manual approach pays dividends in both better performance and simpler semantics. To allow updates to happen at any program point, UpStare’s compiler adds unwinding/rewinding code to all functions; while convenient, this code adds substantial performance overhead to normal execution. Moreover, to exploit UpStare’s flexibility, a developer must carefully define how to map from all possible old-version thread stacks to new-version equivalents. UpStare reduces this burden allowing the programmer to limit updates to fewer program points, just as Kitsune does. But then the value of general-purpose stack reconstruction is less clear. Kitsune allows all compiler optimizations⁵ and code to support control migration imposes no overhead during normal execution since such code only appears on program paths leading to update points, and these paths tend not to intersect with normal execution paths. Moreover, expressing control migration in the code rather than in a specification to the side is arguably advantageous: with only a few update points there is very little code to write, and its presence in the program makes the update semantics explicit and easier to understand. Ekiden, the precursor of Kitsune, effects up-

⁵Compiling the updatable program to use position-independent code (PIC) sacrifices a register. However, modern servers are often compiled with PIC to enable address-space layout randomization.

dates by transferring state to a new-version process; it employs roughly the same API as Kitsune and enjoys its benefits of high flexibility and low overhead, but updates take longer and require more memory.

Data updates. Returning to the table, we can see that most systems handle data structure representation changes by using *object replacement*, in which the programmer, or system, can allocate replacement objects and initialize them using data from the old version. Ksplice and DynaMOS leave the old objects alone but allocate *shadow data structures* that contain only the new fields. Ginseng uses an approach called *type wrapping* wherein programs are compiled so that **structs** have an added version field and extra “slop” space to allow for future growth. Calls to mediator functions are inserted to access updatable objects and these calls initiate transformation of those objects that are not up to date.

Shadow data structures have the benefit that fewer functions are changed by an update: if we add a new field to a **struct**, then only code that uses that field is affected, rather than all code that uses the **struct**. But programmers must write additional code to deal with shadow fields and manage their lifetimes, which imposes run-time overhead and clutters the software over time. Type wrapping has the benefit that there is no need to find objects in order to update them; rather, object transformation will occur lazily as the new version executes. But type wrapping has several limitations: (1) mediator functions slow normal execution; (2) the added slop space hurts performance (e.g., cache locality) and may prove insufficient for some changes; (3) the change in representation forbids certain coding idioms (e.g., involving typecasts to/from **void***); and (4) the whole-program static analysis underlying Ginseng’s type wrapping has trouble scaling.

Object replacement adds the least overhead to normal execution, but there must be a way to find all instances of changed objects (e.g., by chasing pointers from global variables) and redirect these pointers to newly allocated, transformed objects. K42’s coding style makes this easy—the system can just traverse the OTT—but most applications are not written this way. Kitsune’s xfgn tool is able to generate traversal code given relatively small specifications and some type annotations; in other systems, the programmer burden is much higher. Note that DSU for type-safe languages can avoid xfgn’s traversal generation: the garbage collector can automatically find and initiate transformation of changed objects [7, 19] without need of further type annotations.

That said, object replacement in Kitsune is not a panacea. For one, xfgn does not currently support migrating a pointer to the interior of an object before the object itself has been migrated. Type wrapping is not limited in this way because the address of the interior object will not change between versions. However, type wrapping does suffer from a related problem where other data structures may hold onto an address inside a wrapped type (what Neamtiu and Hicks call an *abstraction-violating alias* [17]), which will delay the

update. Another problem with object replacement in Kitsune is that all migration takes place at update time, potentially producing a lengthy pause in execution. Type wrapping postpones transformation of data until it is accessed, which amortizes the transformation cost over the post-update execution. However, type wrapping’s lazy approach could have the undesirable effect of delaying the pause until the application is performing a time-critical request.

Timing. Returning to the table, we consider how systems determine when an update may take effect. Ksplice, K42, and OPUS only permit an update when changed code is not *active*; that is, no thread is running that code, and no thread’s stack refers to it. While this restriction reduces post-update errors, it does not eliminate them [8], and moreover imposes strong restrictions on the form of an update and how quickly it can be applied.

For increased flexibility, other systems allow updates to active code. Kitsune and UpStare updates take place when all threads reach a programmer-designated *update point* (for UpStare, such points may be system-determined). We have found this simple approach works quite well in practice. In contrast, Ginseng allows an update to take effect so long as it *appears* as though it occurred when all threads were at update points [16]. This approach accelerates update times, but the static analysis that underlies it scales poorly and is conservative, requiring awkward code restructurings. POLUS allows threads to update immediately, and thus because POLUS updates take effect at function calls, after an update a program may wind up running bits of old and new code at the same time; a study using Ginseng showed mixing code versions substantially increases the odds of errors [8]. Moreover, POLUS data structures are versioned, with version N of the code accessing version N of the data, so the programmer defines callbacks (invoked via virtual memory page protection support) to keep the copies in sync. Therefore, the programmer must understand the impact of multiple code versions accessing the same data, which we imagine could be tricky when a datastructure change corresponds to a change in semantics. Our experience with the simple barrier approach suggests these more sophisticated approaches, with higher programmer demands, may be unnecessary [11].

Checkpointing. Checkpoint-and-restart systems [18] allow programs to be relaunched “in the middle” of execution from a checkpoint. At a high-level this bears some similarity to DSU, but checkpointing systems do not provide support for changing code or data representations on restart. Ekiden effects an update by serializing and transferring state between an old-version process and a fresh new-version process—i.e., Ekiden works like a checkpointing system that does permit code and data modification. However, we found that the cost of transferring state in Ekiden was significant, and hence moved to the Kitsune model, which has a similar programming API but allows in-place code and data changes, for better performance.

6. Conclusions

We have presented Kitsune, a new system for dynamically updating C programs. Kitsune works by updating the entire program at once, thus avoiding the restrictions imposed by other DSU systems on data representations, programming idioms, and compiler optimizations. Kitsune's design allows program changes for updatability to be simple and informative, and xfgn makes writing state transformers much easier. Our results from applying Kitsune to single- and multi-threaded benchmarks show that Kitsune has essentially no performance overhead, and code changes required to use Kitsune are comparable to, or only slightly more than, prior systems. We believe that the ideas and insights behind Kitsune could also be applied to C++ programs, though extending to `kitc` and `xfgn` to C++ would require non-trivial effort. We believe that Kitsune's careful balancing of flexibility, efficiency, and ease-of-use makes it a major step forward in practical dynamic software updating for C.

Acknowledgments

This research was supported in part by NSF grant CCF-0910530 and the partnership between UMIACS and the Laboratory for Telecommunication Sciences. We would like to thank Karla Saur and Jonathan Turpie for help in the development and testing of Kitsune. Emery Berger, Miguel Castro, JP Martin, Cristi Zamfir, and the anonymous referees provided helpful comments on drafts of this paper. Kristis Makris helped us with the UpStare benchmarks.

References

- [1] G. Altekari, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *Proc. USENIX Security*, 2005.
- [2] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. EuroSys*, 2009.
- [3] A. Baumann, J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing dynamic update in an operating system. In *Proc. USENIX ATC*, 2005.
- [4] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *Proc. VEE*, 2006.
- [5] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew. Dynamic software updating using a relaxed consistency model. *IEEE Transactions on Software Engineering*, 37(5), 2011.
- [6] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. ESOP*, 2007.
- [7] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, 1997. URL <http://www.dcs.ed.ac.uk/home/stg/DynamicML/dynamic.ps.gz>.
- [8] C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster. Evaluating dynamic software update safety using efficient systematic testing. *IEEE Transactions on Software Engineering*, 99(Preliminary), Sept. 2011.
- [9] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster. State transfer for clear and efficient runtime upgrades. In *Proc. HotSWUp*, 2011.
- [10] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proc. International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2012.
- [11] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster. A study of dynamic software update quiescence for multithreaded programs. In *Proc. HotSWUp*, 2012.
- [12] M. Hicks and S. Nettles. Dynamic software updating. *ACM TOPLAS*, 27(6), 2005.
- [13] K. Makris and R. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*, 2009.
- [14] K. Makris and K. D. Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proc. EuroSys*, 2007.
- [15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. ASPLOS*, 2009.
- [16] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI*, 2009.
- [17] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proc. PLDI*, 2006.
- [18] E. Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.
- [19] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proc. PLDI*, 2009.
- [20] ZeroTurnaround. LiveRebel. <http://www.zeroturnaround.com/liverebel>.