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 allowing 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 its semantics easy to understand while keeping programmer work to a minimum. Finally, the programmer can write simple specifications to direct Kitsune to generate code that traverses and transforms old-version state for use by the 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.

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, 3, 7, 9–11, 13, 14, 16]. Concurrently, industry has begun to package DSU support into commercial products [2, 17].

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, e.g., caching servers like Memcached and key-value servers like Redis. 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 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 longjmps back to the driver routine, which loads the new application version and calls the new version's main function. Thus, application 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*, which is the transformation of old global state to be compatible with the new program version; 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 xfgen that makes it easy to write code to migrate and transform old program state to be compatible with a new program version. The input to xfgen is a series of *type and variable transformation specifications*, one per changed type or variable, that describe in intuitive notation how to translate data from the old to new format. The output of xfgen is C code that performs the transformations wherever they are needed: at a high level, the generated transformers operate 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 according to the actions in the transformers, taking care to maintain the shape of the data structures. The old version's copies of any migrated 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 xfgen.)

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 53 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. xfgen was also very effective, allowing us to write state transformers with similarly small specifications consisting of between 27 and 200 lines in total; the size here depends on the number of data structure changes across an application's streak. We tested that all programs behaved correctly under our updates.

We measured Kitsune's performance overhead, and found it ranged from -2.2% to +1.8%, which is in the noise on modern environments [12]. We also found that the time required to perform an update was typically less than 40ms; icecast's longer, ∼1s update time is due to internal timing constraints and does not adversely affect the application. (See Section 4 for full details.)

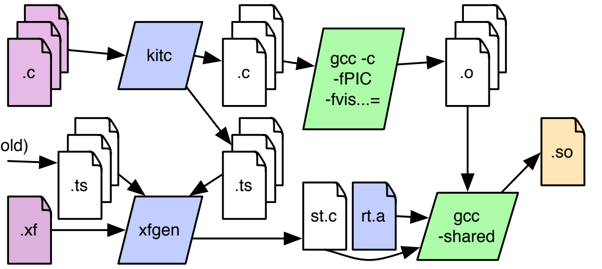

Figure 1. Kitsune build chain

Considered as a whole, we think that Kitsune's design meshes well with C without limiting the form of dynamic updates, and without imposing an undue burden on DSU programmers. In short, we find Kitsune to be the most flexible, efficient, and easy to use (and deploy) DSU system for C developed to date.

2. Kitsune

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 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

[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 worked well in most cases. Only Tor, which had a previously existing control framework, used a different mechanism.

calls the application's (non-exported) main function. The kitsune driver is only 109 lines of C code and is the only part of a program that cannot be dynamically updated.

When SIGUSR2 is received, the handler sets a global flag. As discussed in more detail below, the running program is expected to call the function kitsune_update at points at which an update is permitted to take effect; such calls are dubbed *update points* [9]. 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.

The new program version now must do two things: (1) migrate and transform of 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. We call these activities *data migration* and *control migration*, respectively. The programmer directs the control flow and the timing of state transformation using a few judicious calls into the Kitsune runtime system, and defines state transformation code itself using xfgen.

We next illustrate basic data and control migration using an example and consider xfgen in Section 3.

2.1 Data and Control Migration

The C program in Figure 2 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 needed 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 of the program begins at main() on line 32. After defining some local variables, we call load_config () (code not shown) to initialize the three global configuration variables defined on line 2 and then allocate an empty mapping. Then we call setup_connection () (code also not shown) to begin listening on main_sock, and enter the main loop on lines 43–47. Here we simply wait for a connection and then call 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 there was no command received from the socket, then we exit the loop on line 27.

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 implements Kitsune control and data migration.[2]

```
 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

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

[2] We should emphasize that because this example is tiny, the amount of highlighted code is disproportionately large (see Section 4).

to start each long-running loop. These are good choices for update points because the program is *quiescent*, i.e., in between events, when there is less in-flight state [6, 14].

The kitsune driver will load the new version and call its main function, so the programmer must write code to direct execution back to the equivalent spot in the new program. This code will likely include calls to 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 resumption from normal startup.

In Figure 2, the conditional on line 35 prevents the configuration from being reloaded and mapping from being reallocated when run as an update, since in this case we will migrate that state from the old program version instead (discussed below). If the update was initiated from the client loop, then on line 40 we migrate client_sock from the previous version and then go straight to that loop. Notice that when we return from this call, we will enter the beginning of the main loop, just as if we 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 state. When a Kitsune program starts as an update, critical state from the previous version of the program remains available in memory so it can migrate to the new program version. The programmer is responsible for identifying what state must be migrated, 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 kitsune_note_locals (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 **static**s, 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 *state transformation function* for each registered variable that is automigratable. These functions implement data transformation (versus just copying), and are produced by xfgen from programmer specifications. Each function follows a particular naming convention, and the runtime finds them in the new program version using dlsym(). If no state transformation function is found, the data is copied. xfgen-generated transformers traverse the heap starting from global and (programmer-designated) local variables.

Within a function annotated with kitsune_note_locals , the user calls MIGRATE_LOCAL(var) to migrate (via the appropriate state transformer) the old version of var to the new version, e.g., as used on line 41 to migrate client_sock . MIGRATE_LOCAL() returns 1 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 state 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 all state that might be transformed is reachable from the application's local and global variables. In our experiments, this assumption was only violated in memcached, in which the only pointers to some application data were stored in a library. This problem is addressed by caching such pointers in the main application; see Section 4.1.

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 transformers have marked as freeable. Finally, control returns to the new version.

2.2 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 pthreads program Kitsune-enabled, the programmer must modify 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, the name of thread function f, and the value of f's argument arg are (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 2.1, and continues until 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 initialization 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.

For Kitsune's approach to work, the program must follow several conventions we believe are reasonable. 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 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, and we conjecture adapting non-compliant programs to them should be easy.

3. xfgen

As mentioned briefly in Section 2.1, Kitsune's runtime invokes state transformer functions for each automigrating variable, following a naming convention to locate the appropriate transformer function. In the general case, the developer can construct such functions manually. Kitsune also includes xfgen, a tool that produces state transformation functions from simple specifications. The design of xfgen is based on our experience applying DSU to C [6, 7, 9, 13, 14], and aims to make common kinds of state transformers easy to write while maintaining the flexibility to implement arbitrary transformations.

Figures 3(a) and (b) summarize xfgen's transformer specification language. Each transformer has one of the forms shown in part (a). The INIT transformers describe how instances of new variables or types should be initialized, and the → 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** type field (we will see an example below). The transformer *action* consists of arbitrary C code that may reference the special xfgen variables shown in Figure 3(b), which refer to entities from the old or new program version. A → transformation without an action identifies a variable/type renaming.

We next illustrate the specification language through a series of examples, and then discuss how transformer functions are generated from specifications.

3.1 Example transformers

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. Since we would not know exactly how many of each of these operations had occurred by the time of the update (we only have their sum in op_count), we need to decide how to initialize the new variables. We might determine they should over-approximate the actual count by writing the following xfgen specifications:

```
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.

Example 2. Suppose we change line 5 in Fig. 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:

```
mapping → mapping: {
 int key;
 $out = NULL;
 for(key = 0; key < $oldsym(config_size); key++) {
   if ($in[key] != 0) {
     $newtype(struct list ) *cur =
       malloc( sizeof ($newtype(struct list )));
     cur→ key = key;
     cur→ val = $in[key];
     cur→ next = $out;
     $out = cur;
} } }
```

Here mapping → mapping indicates this is a transformer for the old version of mapping (the occurrence to the left of

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*

(a) transformers

$in, $out – old/new type or var
$old/newsym(x) – x in old/new prog.
$old/newtype(t) – t in old/new prog.
$base – containing **struct**
$xform(old, new) – xformer ref.

(b) special variables

E_PTRARRAY(S) – size of ptd-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

(c) type annotations

Figure 3. xfgen specification language and type annotations

the arrow, referred to as $in within the transformer) to the new version of mapping (referred to as $out). 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 be simply 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. 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 state transformers 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 structurally.)

Other special variables. In the examples so far, we have seen uses of all but the last two special variables in Figure 3(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 refers to a particular type transformer. For example, suppose we merged Examples 2 and 3 into a single update that migrated 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 is passed an old and a new type name (here, both are data) and it looks up (or forces the creation of) the transformer between those types. This transformer is returned as a closure that takes pointers to the old and new object versions and can be called using XF_INVOKE.

3.2 Transformer generation

xfgen generates code to perform migration and transformation from the .xf file and the type summary files (.ts files in Figure 1) of the old and new versions. 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 2.1). xfgen uses type information to generate code that can inspect and manipulate program data, and it uses migration information to make sure .xf files are complete: an .xf file is rejected if it fails to define a transformer for a migratable variable or type that has changed between versions.

Type annotations. xfgen sometimes needs type information beyond what is available in C. For example, suppose we write a transformer for a variable foo *x. Then xfgen needs to know how to traverse the memory pointed to by x, e.g., whether x is a pointer to a single foo instance or an array. In Kitsune, this extra information is provided by the programmer as annotations, shown in Figure 3(c). kitc recognizes these annotations and adds the information supplied by them to the .ts files.

The annotations, inspired by Deputy [4], are fairly straightforward. E_PTRARRAY(S) provides a size S (an integer or variable) for a pointed-to array. E_ARRAY(S) provides a size S for array fields at the end of a **struct** (which can be left unsized in C). E_OPAQUE annotates pointers that should be copied as values, rather than recursed inside during traversals. By default, xfgen assumes that t* values for all types t are annotated with E_PTRARRAY(1); explicit annotations override this default.

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. With the above declaration, we could write **struct** list E_INST(**int**) *x to declare that x is a list of **int**s.

Variable transformers. For each migrated variable listed in the new version's .ts file, if that variable is named explicitly in an old_var → new_var transformer, then xfgen generates C code from the given action, substituting references appropriately. For example, $in and $out in the action 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. 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. xfgen then generates C code that migrates the variable by calling the type transformation from the type of the old-version symbol to x's type (as described next).

As an example, xfgen will produce the following C code from Example 1, above:

```
void _kitsune_transform_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_transform_set_count () { /* as above */ }
```

Type transformers. Generating C code for manually specified type transformers is analogous to what is done for manually specified variable transformers. We also generate transformation functions for all (unchanged) types t of migratable data, i.e., for the following cases: (1) when a migrated variable has type t but no manual transformer (as with **struct** list in Example 3, above); (2) when traversed data references values of type t (e.g., if an unchanged global variable had type **struct** s where s's definition includes a value of type t); and (3) when a transformer for t is referenced directly in a manual transformer (e.g., as with $xform(data, data) mentioned above). For these cases, the functions merely recursively invoke transformation functions on the immediate children of the type in question (skipping NULL values); no values of that type are copied, but pointers may be redirected to values that are.

Whenever a type transformer migrates a pointer, it performs several steps. First, if the pointer is NULL, it does nothing. Otherwise, it checks a global map to see if the pointer has been migrated before; if so it returns the old target. Doing this maintains the shape of the heap and avoids infinite loops during traversal of cycles. If neither of these two conditions apply, it calls the appropriate transformer for the pointer's target. If the pointer is to a global or local variable, it stores the result in the corresponding new-version variable's space. If the pointer target's type has truly changed (and so must have a manual transformer with an action), it mallocs space and stores the result there, remembering the result's address in the global map. It also stores the old-version pointer on a list of addresses to be reclaimed once the update is complete.

Following the above procedure, xfgen will generate transformer/traversal code that will deeply explore the heap and ensure that all pointers to the data and stack segment are transformed to work once the old program is unloaded. If the programmer knows that a particular data structure contains only pointers into the heap (and not to global or local variables) and no pointed-to objects require transformation, she can create transformers that truncate the traversal to reduce update time (we used this trick for redis-mod in Figure 4).

The transformers generated by xfgen assume there are no pointers into the middle of transformed objects. To help check this assumption, we provide an execution mode in which the created transformers use an interval tree to record the start and end of each object they transform. A transformer 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.

4. Experiments

To evaluate Kitsune, we used it to develop updates for five widely deployed server programs. We found that few code changes are required to support updating: between 53 and 159 LOC, the lion's share of which were made to the initial version. Likewise, xfgen specifications were generally small, at around 3-4 lines per changed variable/type. These numbers are comparable to, or better than, prior work. Kitsune performance is uniformly better, with essentially no steady-state overhead. We found that the time required to apply an update ranges from 2ms up to 1s, depending on the program; in all cases, however, the times seem acceptable for typical use.

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 digg.com. *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 would be lost (memcached) or expensive to restore (redis) following an update. Vsftpd also serves

Program	**# Vers**		**LoC**	**Upd**	**Ctrl**	**Data**	**E_∗**	**Oth**	**Σ**	**v→ v**	**t→ t**	**Σ**	**xf LoC**
vsftpd	14	(1.1.0–2.0.6)	12,202	6	26	17+8	6+14	28+8	83+30	9	21	30	101
redis	5	(2.0.0–2.0.4)	13,387	1	2	3	43	4	53	0	4	4	37
Tor	13	(0.2.1.18–0.2.1.30)	76,090	1	39	37+6	19	57	153+6	16	15	31	189
*memcached**	3	(1.2.2–1.2.4)	4,181	4	9	13	20	66	112	12	10	22	27
*icecast**	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

as a useful benchmark because several other DSU systems have used it for evaluation [3, 7, 10, 14].

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.

4.1 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. In all cases, the versions we updated behaved correctly before, during, and after updates were applied.

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, e.g., kitsune_is_updating (), and data migration, e.g., kitsune_do_automigrate(); the number of type annotations for xfgen, e.g., E_PTRARRAY; 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 flow migration and update points were particularly stable, essentially because the top-level control flow structure of the programs rarely changed. We also found control flow migration code relatively easy to write. 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 is a stable program feature that is straightforward to add and easy to maintain.

The rightmost columns of the table describe the xfgen specifications we wrote for each program's updates. In particular, we list the number of variable transformers (v→ v) and type transformers (t→ 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 xfgen code were needed for each transformer. Most state either required no (or very simple) transformation. One of the largest transformations was a 19-line redis rule to choose the right transformation for a **void**∗ field based on an integer key indicating the field's type. When transformation annotations were necessary, they were either obvious (specifying generic types or bounded arrays) or prompted by xfgen.

Now we consider the particulars of each program, and when possible measure the magnitude of these changes against those required by prior systems. In general we find the number of required changes to be comparable.

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, e.g., the connection listener, login processor, command processor, etc.

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, the update point need not be given its own name, but can reuse the name of the update point in the loop that initiated the wrapped call; this is because this loop will reinitiate the call when the update completes.

Other DSU systems. Neamtiu et al. [14] 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 127 LOC changed and 101 lines of state transformation for Kitsune.

Makris and Bazzi [10] 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.

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 extensive use of linked lists and hash tables, and we used xfgen's generics annotations to model their types precisely. The version streak we considered included only code modifications, but we still needed xfgen to migrate data structures that contain global variable addresses (which change with updates). Finally, redis uses custom memory management functions that xfgen does not support, so we modified these functions to directly call the standard `malloc` and `free`. We leave support of custom allocators to future work.

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, and a small number of control flow modifications to prevent reinitialization of updated state. The small size of the latter is particularly surprising given the very large amount of state in Tor. We did need to add code to migrate one object (representing the network consensus) manually, because it cannot be refreshed correctly until the rest of the state has been migrated. The bulk of Tor's changes served to expose DSU functionality in Tor's "control port" interface, e.g., so that updates could be triggered using standard tools.

One challenge was that Tor uses libevent for event processing, and that library stores function pointers inside event **`struct`**s. Tor maintains a list of those **`struct`**s, and we wrote state transformers that update those pointers when a new version is loaded. These transformers, along with similar transformers updating function pointers used by either Tor or the OpenSSL library, comprised the majority of xfgen rules.

We are unaware of prior work applying DSU to Tor.

Memcached. Memcached is a multi-threaded server that uses libevent, like Tor. As with Tor, we needed to make some minor changes to memcached so the updating signal properly reaches the Kitsune library to initiate the update process. We also needed to reinstall new function pointers in libevent after an update. More interestingly, we needed to add code to memcached to maintain a list of active connections, so that xfgen properly generates code to transform these connections at update-time; in the ordinary implementation of memcached, connection objects are not otherwise reachable from global and local variables once they are passed to libevent.

Other DSU systems. Neamtiu and Hicks [13] 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.

Program	Orig (siqr)	Kitsune	Ginseng
64-bit, 4×2.4Ghz E7450 (6 core), 24GB mem, RHEL 5.7			
vsftpd 2.0.6	6.55s (0.04s)	+0.75%	–
memcached 1.2.4	59.30s (3.25s)	+0.51%	–
redis 2.0.4	46.83s (0.40s)	-0.31%	–
icecast 2.3.1	10.11s (2.27s)	-2.18%	–
32-bit, 1×3.6Ghz Pentium D (2 core), 2GB mem, Ubuntu 10.10			
vsftpd 2.0.3	5.71s (0.01s)	+1.79%	+8.05%
memcached 1.2.4	101.40s (0.35s)	-0.49%	+18.44%
redis 2.0.4	43.88s (0.16s)	-1.21%	–
icecast 2.3.1	35.71s (0.68s)	+1.18%	-0.28%

Table 2. Steady-state performance overhead

Icecast. Icecast 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. Thus, it needed more than the usual number of update points. We added annotations to migrate local variables or skip initialization within the entry functions of each thread, as needed. The most complex icecast patch added a new thread to handle authentication (requiring an added update point) and reduced the number of connection threads. Kitsune provides programmatic access to the set of threads during transformation to support these changes.

Other DSU systems. Neamtiu and Hicks [13] also considered updates to the same streak of icecast versions (plus one earlier version). 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.

4.2 Performance

Steady-state performance overhead. We measured the steady-state overhead of Kitsune on all programs except Tor, discussed separately below. For comparison, we also measured the overhead of Ginseng for the three programs (vsftpd, memcached, and icecast) for which Ginseng updates were available.

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. For vsftpd, we measured the time to perform the following interaction 2K times: connect to the server, change directories, and retrieve a directory listing. For icecast, we used a benchmark originally developed for Ginseng [13] that measures the time taken for 16 simultaneous clients to

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

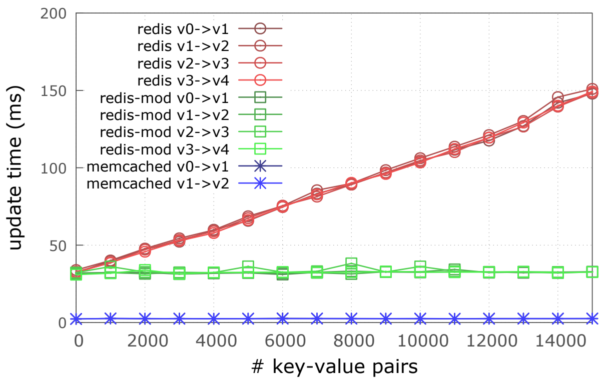

Figure 4. State size vs. update time

download 7 music files, each roughly 2MB in size. For all programs, we ran the client and server on the same machine.

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 and Ginseng (the median Kitsune or Ginseng time 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.

From this data, we can see that Kitsune has essentially no steady-state overhead: the performance differences range from -2.18% to 1.79%, which is well within the noise on modern environments [12]. In contrast, for two of the three programs (vsftpd and memcached), the Ginseng overhead is more significant. While we have not ourselves benchmarked UpStare, the authors of that system report vsftpd overheads of 4.9% and 7.4%, depending on the features enabled [10].

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 0.2.1.28 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, i.e., the elapsed time from when an update is signaled as available to when the update 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 workload 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 1K and 15K 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 [8], 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 xfgen-generated transformers 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 state. Figure 4 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

	Code upd			Data upd			Timing	
	tramp	ind	prog	repl	shdw	wrap	nonactv	upd pts
DynaMOS[4]	×				×			
Ekiden			×	×				×
Ginseng[1,2,3,4,5]		×				×		×
K42[2,5]		×		×			×	
Kitsune			×	×				×
Ksplice	×				×		×	
OPUS[2]	×			–	–	–	×	
POLUS[2,4,5]	×			×				
UpStare[2,3]			×	×				(×)

[1]needs deep analysis
[2]inhibits optimizations
[3]pervasive instrumentation
[4]mixes old and new code
[5]relaxed thread sync.

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

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

Table 4 characterizes the mechanisms used to implement Kitsune and other recent C/C++ DSU systems; most of these systems target applications, while Ksplice, K42, and DynaMOS support (or are) OS kernels. 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.

Code updates. Most systems effect code updates at the level of individual functions (or objects). As noted in the first column, Ksplice [2], OPUS [1], DynaMOS [11], and POLUS[3] [3] insert, at run-time, a trampoline in the old function to jump to the function's new version. As noted in the second column, Ginseng [14] 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 more 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 ("steady state") execution and also inhibits inlining. Most onerously, neither trampolines nor indirections support updating functions that never exit, such as main, which changes relatively frequently [6], or the scheduling loop in the OS. In the best case, programmers must refactor the program so that long-running loop bodies are in separate functions [14].

The remaining three systems, UpStare [10], Ekiden [7],[4] 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 *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 imposes performance overhead on 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 optimizations, 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.

Data updates. Returning to the table, we can see that most systems handle changes to data structure representations employing *object replacement*, in which the programmer 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*: programs are compiled to use mediator functions to access updatable objects, and these functions initiate transformation of 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 man-

[3] LUCOS is from the same research group that produced POLUS, and is essentially a version of POLUS that uses VMMs to effect changes in operating systems. All comments we make about POLUS apply equally to LUCOS, so we do not mention it further in this section.

[4] Ekiden is the precursor of Kitsune and works by transferring state to an updated process; the programmer API is roughly the same, but Ekiden induces slower update times and requires more memory.

age 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) objects must be compiled to have extra "slop" space for future growth which 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**$*$), which Ginseng identifies using a whole-program analysis that has trouble scaling.

Object replacement offers the best steady state performance, 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 xfgen tool is able to generate traversal code given relatively small specifications and some type annotations; in other systems, the programmer burden is higher. Note that DSU for type-safe languages can avoid xfgen's traversal generation: the garbage collector can be used to automatically find and initiate transformation of changed objects [5, 16] without need of further type annotations.

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 no thread is running code that will be changed. While this restriction reduces post-update errors, it does not eliminate them [6], 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 [13]. This approach does accelerate 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 [6]. 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. We imagine this could be tricky. Our experience with the simple barrier approach suggests these more sophisticated approaches, with higher programmer demands, may be unnecessary.

Checkpointing. Checkpoint-and-restart systems [15] 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. As mentioned above, in earlier work we developed Ekiden, a system that serializes and transfers state between processes—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 allows in-process code and data changes.

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 xfgen makes writing state transformers much easier. Our results 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 xfgen 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, and Cristi Zamfir provided helpful comments on drafts of this paper.

References

[1] G. Altekar, 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] 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), Sept. 2011.

[4] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. ESOP*, pages 520–535, 2007.

[5] 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`.

[6] 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(PrePrints), Sept. 2011.

[7] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster. State transfer for clear and efficient runtime upgrades. In *Proc. HotSWUp*, 2011.

[8] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster. A study of dynamic software update quiescence in multi-threaded programs, 2012. Under Review.

[9] M. Hicks and S. Nettles. Dynamic software updating. *ACM TOPLAS*, 27(6), 2005.

[10] K. Makris and R. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*, 2009.

[11] K. Makris and K. D. Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proc. EuroSys*, 2007.

[12] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. ASPLOS*, pages 265–276, 2009.

[13] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI*, 2009.

[14] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proc. PLDI*, 2006.

[15] E. Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.

[16] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proc. PLDI*, 2009.

[17] ZeroTurnaround. LiveRebel. `http://www.zeroturnaround.com/liverebel`.