Practical Dynamic Software Updating for C[*]

Iulian Neamtiu, Michael Hicks
University of Maryland
{neamtiu,mwh}@cs.umd.edu

Gareth Stoyle
University of Cambridge
Gareth.Stoyle@cl.cam.ac.uk

Manuel Oriol
ETH Zurich
moriol@inf.ethz.ch

CS-TR-4790, UMIACS-TR-2006-14

Abstract

Software updates typically require stopping and restarting an application, but many systems cannot afford to halt service, or would prefer not to. *Dynamic software updating* (DSU) addresses this difficulty by permitting programs to be updated while they run. DSU is appealing compared to other approaches for on-line upgrades because it is quite general and requires no redundant hardware. The challenge is in making DSU *practical*: it should be flexible, and yet safe, efficient, and easy to use.

In this paper, we present a DSU implementation for C that aims to meet this challenge. We compile programs specially so that they can be dynamically patched, and generate most of a dynamic patch automatically. Our compiler performs a series of analyses that when combined with some simple runtime support ensure that an update will not violate type-safety while guaranteeing that data is kept up-to-date. We have used our system to construct and dynamically apply patches to three substantial open-source server programs—*Very Secure FTP daemon*, *OpenSSH daemon*, and *GNU Zebra*. In total, we dynamically patched each program with three years' worth of releases. Though the programs changed substantially, the majority of updates were easy to generate. Performance experiments show that all patches could be applied in less than 5 ms, and that the overhead on application throughput due to updating support ranged from 0 to at most 32%.

1 Introduction

Many systems require continuous operation but nonetheless must be updated to fix bugs and add new features. For ISPs, credit card providers, brokerages, and on-line stores, being available 24/7 is synonymous with staying in business: an hour of downtime can cost hundreds of thousands, or even millions of dollars [26, 28]. Many more systems would *prefer* on-line upgrades in lieu of having to stop and restart the system every time it must be patched; an obvious example is the personal operating system. In a large enterprise, such reboots can have a large administrative cost [35]. Despite this, stop/restart upgrades are common—one study [22] found that 75% of nearly 6000 outages of high-availability applications were planned for hardware and software maintenance.

In prior work, we and others have proposed variations of a fine-grained, compiler-based approach to supporting on-line upgrades which we call *dynamic software updating* (DSU). In this approach, a running program is patched with new code and data on-the-fly, while it runs. DSU is appealing because of its generality: in principle any program can be updated in a fine-grained way. There is no need for redundant hardware or special-purpose software architectures, and application state is naturally preserved between updated versions, so that current processing is not compromised or interrupted. DSU can also be used naturally to support dynamic profiling, debugging, and "fix-and-continue" software development. Nonetheless, there has been little implementation experience reported in the literature to suggest that DSU can work in practice for non-stop services written in mainstream programming languages. (A review of past work appears in Section 8.)

This paper presents Ginseng, a new DSU implementation for C programs that aims to satisfy three criteria we believe are necessary for practicality:

DSU should not require extensive changes to applications. DSU should permit writing applications in a natural style: while an application writer should anticipate that software will be upgraded, he should not have to know what form that update will take.

[*]Extended version of *Practical Dynamic Software Updating for C* appearing in Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, June 11–14, 2006, Ottawa, Ontario, Canada

DSU should restrict the form of dynamic updates as little as possible. The power and appeal of DSU is to permit applications to change on the fly at a fine granularity. Thus, programmers should be able to change data representations, change function prototypes, reorganize subroutines, etc. as they normally would.

Dynamic updates should be neither hard to write nor hard to establish as correct. The harder it is to develop applications that use DSU, the more its benefit of finer granularity and control is diminished.

To evaluate Ginseng, we have used it to dynamically upgrade three open-source servers: `vsftpd` (the Very Secure FTP daemon), the `sshd` daemon from the OpenSSH suite, and the `zebra` server from the GNU Zebra routing software package.

Based on our experience, we believe Ginseng squarely meets the first two criteria for the class of single-threaded server applications we considered, and makes significant headway toward the third. These programs are realistic, substantial, and in common use. Though they were not designed with updating in mind, we had to make only a handful of changes to their source code to make them safely updateable. Each dynamic update we performed was based on an actual release, and for each application, we applied updates corresponding to at least three years' worth of releases, totaling as many as twelve different patches in one case. To achieve these results, we developed several new implementation techniques, including new ways to handle the transformation of data whose type changes, to allow dynamic updates to infinite loops, and to allow updates to take effect in programs with function pointers. Though we have not optimized our implementation, overhead due to updating is modest: between 0 and 32% on the programs we tested.

Despite the fact that changes were non-trivial, generating and testing patches was relatively straightforward. We developed tools to generate most of a dynamic patch automatically by comparing two program versions, reducing programmer work. More importantly, Ginseng performs two safety analyses to determine times during the running program's execution at which an update can be performed safely. The theoretical development of our first analysis, called the *updateability analysis*, is presented in earlier work [33]. The contribution of this paper is the implementation of that analysis for the full C programming language, along with some practical extensions, and the development of a new *abstraction-violating alias analysis* for handling some of the low-level features of C. These safety analyses go a long way toward ensuring correctness, though the programmer needs a clear "big picture" of the application e.g., interactions between components and global invariants.

In short, we make the following contributions in this paper:

1. We present a practical framework to support dynamically updating running C programs. Ours is the most flexible, and arguably the most safe, implementation of a DSU system to date.

2. We present a substantial study of the application of our system to three sizeable C server programs. Our experience shows that DSU can be practical for updating realistic server applications as they are written now, and as they evolve in practice. We are optimistic that our approach can ultimately be practical for many non-stop applications, including game servers, operating systems and embedded systems software.

The next section presents an overview of our approach and outlines the rest of the paper.

2 Ginseng Overview

Ginseng consists of a compiler, a patch generator and a runtime system for building updateable software.[1] Basic usage is illustrated in Figure 1, with Ginseng components in white boxes. There are two stages. First, for the initial version of a program, $\texttt{v}_0\texttt{.c}$, the *compiler* generates an updateable executable $\texttt{v}_0$, along with some prototype and analysis information (`Version Data` $\texttt{d}_0$). The executable is deployed. Second, when the program has changed to a new version ($\texttt{v}_1\texttt{.c}$), the developer provides the new and old code to the *patch generator* to generate a patch $\texttt{p}_1\texttt{.c}$ representing the differences. This is passed to the compiler along with the current version information, and turned into a `dynamic patch` $\texttt{v}_0 \rightarrow \texttt{v}_1$. The *runtime system* links the dynamic patch into the running program, completing the on-line update. This process continues for each subsequent program version.

The Ginseng compiler has two responsibilities. First, it compiles programs to be dynamically updateable, so that existing code will be redirected to replacement functions present in a dynamic patch. In addition, when a type is updated, existing values of that type must be transformed to have the new type's representation, to be compatible with the new code. Code is compiled to notice when a typed value is out of date, and if so, to apply the necessary

[1]The compiler and patch generator are written in Objective Caml using the CIL framework [25].The runtime system is a library written in C.

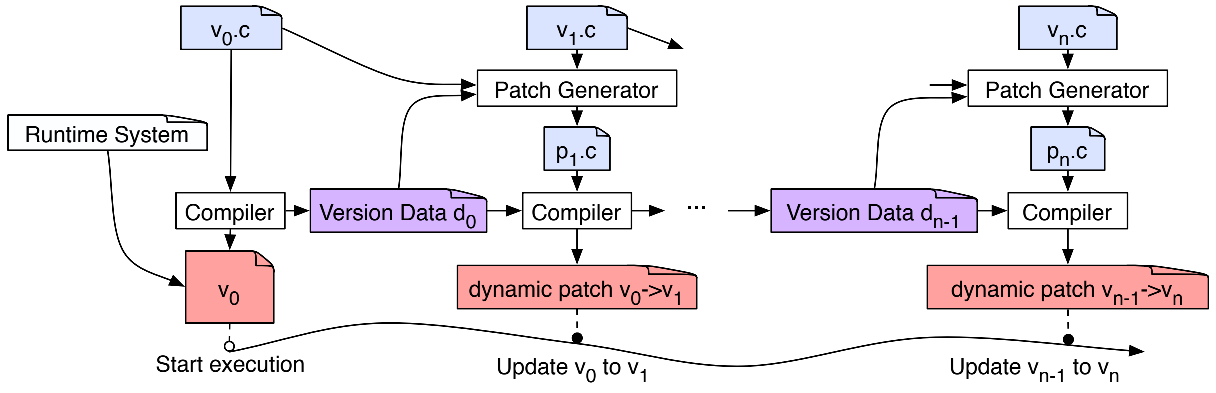

Figure 1: Building and dynamically updating software with Ginseng.

transformation function. We explain in Section 3 how our implementation supports these features by transforming a program to use *function indirections* and *type wrappers*.

Second, the Ginseng compiler uses a suite of analyses to ensure that updates are always *type-safe*, even when changes are made to function prototypes or type definitions. The basic idea is to examine the program to discover assumptions made about the types of updateable entities (i.e., functions or data) in the continuation of each program point. These assumptions become constraints on the timing of updates. For example, a call to `int f(int)` constrains the program point just before the call to not allow an update to `f` that would change `f`'s type. The formal details of our analysis are presented elsewhere [33]; Section 4 discusses its application to C programs, including several extensions.

The Ginseng patch generator (Section 5) has two responsibilities. First, it identifies those definitions (be they global variables, functions, or types) that have changed between versions. New and changed definitions are included in the output patch file, while old definitions are made `extern`. Second, for each type definition that has changed, it generates a *type transformer* function used to convert values from a type's old representation to the new one. The compiler inserts code so that the program will make use of these functions following a dynamic patch. If the new code assumes an invariant about global state, this invariant has to hold after the update takes place. Users can write optional *state transformer* functions that are run at update time to convert global state and run initialization code for this purpose. Users also may adjust the generated type transformers as necessary. We found that writing state transformers or adjusting type transformers was rarely needed.

The dynamic update itself is carried out by the Ginseng runtime system (Section 5) linked into the updateable program. Once notified, the runtime system will cause the patch to be dynamically loaded and linked at the next safe *update point*. This is essentially a call into the runtime system inserted by the programmer. Our safety analysis will annotate these points with constraints as to how definitions are allowed to change at each particular point. The runtime system will check that these constraints are satisfied by the current update, and if so, it "glues" the dynamic patch into the running program. In our experience, finding suitable update points in long-lived server programs is quite straightforward, and the analysis provides useful feedback as to whether the chosen spots are free from restrictions.

The next three sections describe these features of Ginseng in detail, while Sections 6 and 7 describe our experience using Ginseng and evaluate its performance. We finish with a discussion of related work and conclude.

3 Enabling On-line Updates

To make programs dynamically updateable we address two main problems. First, existing code must be able to call new versions of functions, whether via a direct call or via a function pointer. Second, the state of the program must be transformed to be compatible with the new code. For a type whose definition has changed, existing values of that type must be transformed to conform to the new definition.

Ginseng employs two mechanisms to address these two problems, respectively: *function indirection* and *type-wrapping*. We discuss them in turn below, and show how they can be combined to update long-running loops.

3.1 Function Indirection

Function indirection is a standard technique that permits old code to call new function versions by introducing a level of indirection between a caller and the called function, so that its implementation can change. For each function `f` in the program, Ginseng introduces a global variable `f_ptr` that initially points to the first version of `f`.[2] Ginseng encodes version information through name mangling, `f` initially being `f_v0`, then `f_v1` and so on. Each direct call to `f` within the program is replaced with a call through `*f_ptr`. Ginseng also handles function pointers in an interesting way: if the program passes `f` as data (i.e., as a function pointer), Ginseng generates a wrapper function that calls `*f_ptr` and passes this wrapper instead. To dynamically update `f` to version 1, the runtime system dynamically loads the new version `f_v1` and then stores the address of `f_v1` in `f_ptr`.

3.2 Type Wrapping

The Ginseng updating model enforces what we call *representation consistency* [33], in which all values of type `T` in the program at a given time must logically be members of `T`'s most recent version. The alternative would be to allow multiple versions of a type to coexist, where code and values of old and new type could interact freely within the program. (Hjálmtýsson and Gray [18] refer to these approaches as *global update* and *passive partitioning*, respectively.) Representation consistency is a useful property because it more closely models the "forward march" of a program's on-line evolution, making it easier to reason about.

To enforce representation consistency, Ginseng must ensure that when a particular type `T`'s definition is updated, values of that type in the running program are updated as well. To do this, a dynamic patch defines a *type transformer function* used to transform a value $v_{\mathtt{T}}$ from `T`'s old definition to its new one. Just like functions, types are associated with a version, and the type transformer $c_{\mathtt{T}_{n \to n+1}}$ converts values of type $\mathtt{T}_n$ to be those of type $\mathtt{T}_{n+1}$. As we explain later, much of a type transformer function can be generated automatically via a simple comparison of the old and new definitions.

Given this basic mechanism, we must address two questions. First, when are type transformers to be used? Second, how is updateable data represented?

Applying Type Transformers To transform existing $v_{\mathtt{T}_n}$ values the runtime system must find them all and apply $c_{\mathtt{T}_{n \to n+1}}$ to each. One approach would be to do this eagerly, at update-time; this would require either implementing a garbage-collector-style tracing algorithm [14], or maintaining a registry of pointers to every (live) value of type $\mathtt{T}_n$ during execution [4]. More simply, we could restrict type transformation to only those data reachable from global variables, and require the programmer to implement the tracer manually [17]. Finally, we could do it lazily, as the program executes following the update [12, 7].

Ginseng uses the lazy approach. The compiler renames version n of the user's definition of `T` to be $\mathtt{T}_n$, where the definition of `T` simply wraps that of $\mathtt{T}_n$, adding a `version` field. Given a value $v_{\mathtt{T}}$ (of wrapped type `T`), Ginseng inserts a *coercion* function called $con_{\mathtt{T}}$ (for $\underline{\text{con}}$cretization of `T`) that returns the underlying representation. This coercion is inserted wherever $v_{\mathtt{T}}$ is used concretely, i.e., in a way that depends on its definition. For example, this would happen when accessing a field in a `struct`. Whenever $con_{\mathtt{T}}$ is called on $v_{\mathtt{T}}$, the coercion function compares $v_{\mathtt{T}}$'s version n with the latest version m of `T`. If $n < m$, then the necessary type transformer functions are composed and applied to $v_{\mathtt{T}}$, changing it in-place, to yield the up-to-date $v_{\mathtt{T}_m}$ (of type $\mathtt{T}_m$).

The lazy approach has a number of benefits. First, it is not limited to processing only values that are reachable by global variables; stack-allocated values, or those reachable from stack allocated values are handled easily. Second, it amortizes transformation costs, reducing the potential pause at update-time that would be required to transform all data in the program. The drawback is that per-type access during normal program execution is more expensive (due to the calls to $con_{\mathtt{T}}$), and the programmer has little control over when type transformers are invoked, since this is determined automatically. Therefore, transformers must be written to be timing-independent. In our experience, type transformers are used rarely, and so it may be sensible to use a combination of eager and lazy application to reduce total overhead.

Without care, it could be possible for a transformed value to end up being processed by old code, violating representation consistency. This could lead a $con_{\mathtt{T}}$ coercion to discover that the version n on $v_{\mathtt{T}}$ is actually *greater* than the version m of the type `T` expected by the code. A similar situation arises when function types change: old code might end up calling the new version of a function assuming it has the old signature. We solve these problems with some novel safety analyses, described in more detail in Section 4.

[2]Ginseng is more careful than we are in these examples about generating non-clashing variable names.

Original program

```
struct T {
  int x; int y;
};

void foo(int* x) {
  *x = 1;
}
void apply(void (*fp)(int*),
           int* x) {
  fp(x);
}
void call() {
  struct T t = {1,2};
  apply(foo,&t.x);
  t.y = 1;
}
```

Updateable program

```
struct T {
  unsigned int version;
  union { struct __T0 data;
          char padding[X]; } udata;
};
struct __T0* __con_T(struct T* abs) {
  __DSU_transform(abs);
  return &abs->udata.data;
}

void * foo_ptr = &__foo_v0;
void * apply_ptr = &__apply_v0;
void * call_ptr = &__call_v0;

void __foo_wrap(int* x) {
  (*foo_ptr)(x);
}
```

```
struct __T0 { int x; int y; };
/* D=D'={T}, L={T}, x:T */
void __foo_v0(int* x) { *x = 1; }
/* D={foo,T}, D'={T}, L={}, x:T */
void __apply_v0(void (*fp)(int*),
                int *x) {
  fp(x);
}
/* D={T,apply}, D'={}, L={} */
void __call_v0() {
  struct T t = { 0, {.data={1,2}}};
  (*apply_ptr)(__foo_wrap,
               &(__con_T(&t))->x);
  /* D={T} */
  &(__con_T(&t))->y = 1;
}
```

Figure 2: Compiling a program to be dynamically updateable.

Type Representations While lazy type updating is not new, there has been little or no exploration of its implementation, particularly for a low-level language such as C. Based on our experience, a given type is likely to grow in size over time, so the representation of the wrapped type T must accommodate this. One approach is to define the wrapper type to use a fixed space, larger than the size of T_0 (padding). This strategy allows future updates to T that do not expand beyond the preallocated padding. The main advantage of the padding approach is that the allocation strategy for wrapped data is straightfowrward: stack-allocated data in the source program is still stack allocated in the compiled program, and similarly for `malloc`ed data. This is because type transformation happens *in place*: the transformed data overwrites the old data in the same storage. On the other hand, growth in the size of a data type is limited by the initial padding, hampering on-line evolution. Padding also changes the cache locality of data; for example, if a two-word structure in the original program is expanded to four words, then half as many elements can fit in a cache line. For simplicity, Ginseng employs this approach.

An alternative approach is to use indirection, and represent the wrapped type as a pointer to a value of the underlying type. This mechanism is used in the K42 operating system [20], which supports updating objects. The indirection approach solves the growth problem by allowing the size of the wrapped type to grow arbitrarily, but introduces an extra dereference per access. More importantly, the indirection approach makes memory management more challenging: how should storage for the transformed data be allocated, and what is to happen to the now-unneeded old data? Also, when data is copied, the indirected data must be copied as well, to preserve the sharing semantics of the application. The simplest solution would be to have the compiler `malloc` new representations and `free` (or garbage collect) the old ones; this is less performance-friendly than stack allocation. A better alternative would be to use *regions* [34], which have lexically-scoped lifetimes (as with stack frames), but support dynamic allocation. Of course, a hybrid approach is also possible: data could start out with some padding, and an indirection is only added if the padding is ever exceeded.

3.3 Example

Figure 2 presents a simple C program, and how we compile it to be updateable. The original program is on the left, and the resulting updateable program in the middle and right columns. The comments can be ignored; these are the results of the safety analysis, explained in the next section.

First, we can see that all function definitions have been renamed to include a version, and that Ginseng has introduced a `_ptr` variable for each function, to keep a pointer to the most current version. Calls to functions are indirected through these pointers. Second, we can see that the definition of `struct T` is now a wrapper for `struct __T0`, the original definition. The `__con_T` function unwraps a `struct T`, potentially transforming it first via a call to `__DSU_transform`. The `__con_T` function is called twice in `__call_v0` to extract the underlying value of `t`. Finally, we can see that Ginseng has generated `__foo_wrap` to wrap an indirected call to `foo`; this is passed as a function pointer to `apply`.

Original program

```
int foo(float g) {
  int x = 2;
  int y = 3;
  L1:while (1) {
    x = x+1;
    if (x == 8) break;
    else continue;
    if (x == 9) return 42;
  }
  return 1;
}
```

Updateable program

```
struct L1_ls {
  float *g;  int *x;  int *y;
};

int L1_loop(int *ret,
              struct L1_ls *ls) {
  *(ls->x) = *(ls->x) + 1;
  if (*(ls->x) == 8) {
    return (0); // break
  } else {
    return (1); // continue
  }
  if (*(ls->x) == 9) {
    *ret = 42;
    return (2); // return
  }
  return (1);   // implicit continue
}
```

```
int foo(float g) {
  int x = 2;
  int y = 3;
  struct L1_ls ls;
  int retval;
  int code;
  ls.g = & g; // init loop state
  ls.x = & x;
  ls.y = & y;
  while (1) {
    code = L1_loop(&retval, &ls);
    if (code == 0) break;
    else if (code == 1) continue;
    else return (retval);
  }
  return (1);
}
```

Figure 3: Loop extraction.

3.4 Loops

When a function f is updated, in-flight calls are unaffected, but all subsequent calls, including recursive ones, take the new f. In general, this is a good thing, because it makes reasoning about the timeline of an update simpler. On the other hand, this presents a problem for functions that implement long-running or infinite loops: if an update occurs to such a function while the old version is active, then the new version may not take effect for some time, or may never take effect.

We solve this problem by a novel transformation we call *loop extraction*. The idea is that the body of a loop can be extracted into a separate function that is called on each iteration of the loop. If the function containing the loop is later changed, then this extracted function will notice the changes to the loop on the next iteration. As the code and state preceding the loop might have changed as well, the loop function must be parameterized by some *loop state*. This state will be transformed using our standard type transformer mechanism on the next iteration of the loop. Extracting the loop body into a function parameterized by loop state is similar to closure conversion followed by lifting.

For illustration, consider the code in the left column of Figure 3. The programmer directs Ginseng that the loop labeled L1 should be extracted. The result is shown in the middle and right columns. In the middle is the extracted loop function, L1_loop, and on the right side is the rewritten original function foo. The function L1_loop takes two arguments: struct L1_ls *ls, and int *ret. The first argument is the loop state, which contains pointers to all of the local variables and parameters referenced in foo that might be needed by the loop; we can see in foo where this value is created. Within L1_loop, references to local variables (x) or parameters (g) have been changed to refer to them through *(ls).

Within the function foo, the loop function is called on each loop iteration. Within the extracted loop function, expressions that would have exited the loop—notably break, continue, and return statements—are changed to return x, where x is 0 for break, 1 for continue and 2 for return. In foo, this return code is checked and the correct action is taken.

If in a subsequent program version the loop in foo were to change, the extracted versions of the two loops would be different, with the new one updating the old one. The new version will be invoked on the loop's next iteration, and if the new loop requires additional state (e.g., new local variables or parameters were added to foo), then this is handled by the type transformer function for struct L1_ls. This type transformer might perform side-effecting initialization as well, for code that would have preceded the execution of the current loop. Note that foo's callers are neither aware nor affected by the loop extraction inside the body of foo.

When extracting infinite loops, nothing else needs to be done. However, if the loop might terminate, we must extract the code that follows the loop as well, so that an updated loop does not execute a stale postamble when it completes. This can be done using loop extraction itself: to extract a statement S, the programmer rewrites that statement to be `while (1){` S; `break;}`, and then Ginseng extracts the loop.

Replacing arbitrary code on the stack was critical for supporting two of our three benchmark applications, `vsftpd` and `sshd`(Section 6). Both applications are structured around event loops: a parent process accepts incoming connection requests, and forks. The forked child breaks out of the loop and executes the loop postamble. If the loop body and loop postamble change in later versions, this will translate into updates to both extracted functions, hence both the parent and the children will get to execute the most up to date version.

4 Safety Analysis

Let us look again at the example in Figure 2. Suppose the program has just entered the `call` function—is it safe to update the type `T`? Generally speaking the answer is no, because code `t.x` assumes that `t` is a structure with field `x`, and a change to the representation of `t` could violate this assumption, leading to unexpected behavior. In this section we look at how Ginseng helps the programmer avoid choosing bad update points like this one using static analysis.

4.1 Tracking Changes to Types

The example given above illustrates what could happen when old code accesses new data, essentially violating representation consistency. To prevent this situation from happening, Ginseng applies a constraint-based, flow-sensitive *updateability analysis* [33] that annotates each update point with the set of types that may not be updated if representation consistency is to be preserved. This set is called the *capability* because it defines those types that *can* be used by old code that might be on the call stack during execution. Of course, the capability is a conservative approximation, as it approximates all possible "stack shapes." It is computed by propagating concrete uses of data backwards along the control flow of the program to possible update points.

Statically-approximated capabilities are illustrated in Figure 2, where the sets labeled D in the comments define the current capability; on functions, D defines the capability at the start of the function and D' defines it at the end. When `T` appears in D, it means that the program has the *capability* to use data of type `T` concretely. An update must not revoke this capability when it is needed. For example, the concrete use of `t` at the end of the `call` function requires `T` to be in D, which in turn forces `apply` not to permit an update to `T`.

Programmers indicate where updates may occur in the program text by inserting a call to a special runtime system function `DSU_update`. When our analysis sees this function, it "annotates" it with the current capability. At run-time this annotation is used to prevent updates that would violate the static assumption of the analysis. Moreover, the runtime system ensures that if a type *is* updated, then any functions in the current program that use the type concretely are updated with it. This allows the static analysis to be less conservative. In particular, although the constraints on the form of capabilities induced by concrete usage are propagated backwards in the control flow, propagation does not continue into the callers of a function. This propagation is not necessary because the update-time check ensures that all function calls are always compatible with any changed type representations.

We have formalized the updateability analysis and proved it correct in previous work [33]. One contribution of the present work is the implementation of this analysis for the full C language. Our implementation extends the basic analysis to also track concrete uses of functions and global variables, which permits more flexible updates to them. In the former case, by considering a call as a concrete use of a function, and function names as types, we can safely support a change to the type of the function. Similarly, in the latter case, by taking reads and writes of global variables as concrete uses, and the name of a global variable as a type, we can support representation changes to global variables. In our experience, the types of functions and global variables do change over time, so this extension has been critical to making DSU work for real programs.

To illustrate the analysis, consider Figure 2 again. We can see that function names appear in the initial capability of `apply` and `call`. In the former case, this is because the analysis determines that `fp` could be `foo` at run-time, and thus the call to `fp` places `foo` (and other functions `fp` could be) into the current capability. For the latter case, the call to `apply` within `call` places it in `call`'s initial capability. This means that if we were to attempt an update at the start of `apply` (respectively `call`), then the type of `foo` (respectively `apply`) must either remain unchanged or the new type be a subtype of the old type [33].

The implementation also properly accounts for both signals and non-local control transfers via `setjmp`/`longjmp`, albeit quite conservatively. Since signal handlers can fire at any point in the program, we prevent updates from occurring inside a signal handler (or any function that handler might call), to avoid violating assumptions of the analysis (we could allow updates to occur, but prevent updates that would change type representations, function signatures, etc.) We model `setjmp`/`longjmp` as non-local `goto`; that is, the updateability analysis assumes that any `longjmp` in the program could go to any `setjmp`. The server programs in Section 6 do not employ `setjmp`/`longjmp`, but all of them use signals.

In future work, we plan to extend our approach to multithreaded programs. Because thread executions are interleaved, we will have to either extend our safety analysis to account for capabilities of other threads, and/or synchronize threads at safe update points before allowing an update to take effect [31].

4.2 Abstraction-Violating Aliases

C's weak type system and low level of abstraction sometimes make it difficult for us to maintain the illusion that a wrapped type is the same as its underlying type. In particular, the use of unsafe casts and the address-of (`&`) operator can reveal a type's representation through an alias. An example of this can be seen in Figure 2 where `apply` is called passing the address of field `x` of struct `t`. Within `foo`, called by `apply` with this pointer, the statement `*x = 1` is effectively a concrete use of `T`, but this fact is not clear from `x`'s type, which is simply `int *`. An update to the representation of `struct T` while within `foo` could lead to a runtime error. We have a similar situation when using a pointer to a `typedef` as a pointer to its concrete representation. We say that these aliases are *abstraction violating*.

One extreme solution would be to mark `structs` whose fields have their address taken as non-updateable. However, this solution can be relaxed by observing that only as long as an alias into a value of type `T` exists is it dangerous to update `T`. Thus if we know, at each possible update point, those types whose values might have live *abstraction-violating aliases* (AVAs), we can prevent those types from being changed.

We discover this set of types using a novel *abstraction violating alias analysis*. The analysis follows the general approach of effect reconstruction [23, 10, 1], and is described in more detail in Stoyle's thesis [32]. Pointers are annotated with an "effect" which lists the types whose values they may be pointing into. For example, a pointer created by `&t.x` would include the type of `t` in its effect. If such a pointer might be live at an update point, then no types in its effect may be updated. To approximate the set of live pointers at a given program point, we simply need to look to the lexical environment of the program at that point, along with the lexical environments of possible callers to the current function, ultimately back to `main()`. For each function, we define a set L as those types with abstraction violating pointers in at least one of the callers' environments. We calculate this set through a simple constraint based analysis that uses the control flow of the program. Finally, the capability of each possible update point is extended to include the current function's L and the effects appearing in the free variables of the current environment.

The comments in Figure 2 illustrate the AVA analysis results for the example, where L's contents are shown for each function, and the effect associated with variable `x` in functions `foo` and `apply` is shown to be `T` via the notation `x:T`. Looking at the example, we can see the `call` function violates `T`'s abstraction by taking the address of `t.x`, and then passes this pointer to `apply`. This pointer is not used concretely in `call`, so does not effect subsequent computation in this function: `call`'s environment has no abstraction violating pointers. As `call` is the only caller of `apply`, its associated L is empty. However, the environment of the body of `apply` does contain an abstraction-violating pointer, namely the parameter `x`. Thus when `apply` calls `foo` via the pointer `fp`, `T`'s abstraction is violated and the L annotation for `foo` must contain `T`. In the example, we consider all statements as possible update points, and so extend D according to the results of the AVA analysis. This is why, for example, `T` appears in the capability of both `foo` and `apply`. In both cases `T` is in L or in the effect of a free variable in the environment (i.e., `x`).

4.3 Unsafe Casts and `void *`

To ensure that the program operates correctly, many representation-revealing casts are disallowed. For example, if we had a declaration `struct S { int x; int y; int z; }`, a C programmer might use this as a subtype of `struct T` from Figure 2, by casting a `struct S *` to a `struct T *`. Given the way that we represent updateable types, permitting this cast would be unsafe, since `struct S` and `struct T` might have distinct type transformers and version numbers and treating one as the other may result in incorrect transformation. As a result, when our analysis discovers such a cast, it rules both types as non-updateable.

However, it would be too restrictive to handle all such casts this way. For example, C programmers often use `void *` to program generic types. One might write a "generic" container library in which a function to insert an element takes a `void *` as its argument, while one that extracts an element returns a `void *`. The programmer would cast the inserted element to `void *` and the returned `void *` value back to its assumed type. This idiom corresponds to *parametric polymorphism* in languages like ML and Haskell. Programmers also encode *existential types* using `void *` to build constructs like callback functions, and use upcasts and downcasts when creating and using callbacks, respectively.

If these idioms are used correctly, then they pose no problem to Ginseng's compilation approach since they do not reveal anything about a type's representation. However, we cannot treat casts to and from `void *` as legal in general, because `void *` could be used to "launder" an unsafe cast. For example, we might cast `struct S *` to `void *`, and then the `void *` to `struct T *`. Each cast may seem benign on its own, but becomes unsafe in combination. To handle this situation, our analysis annotates each `void *` type in the program with the set of concrete types that might have been cast to it, e.g., casting a `struct T *` to a `void *` would add `struct T` to the set. When casting a `void *` to `struct S *`, the analysis ensures the annotation on the `void *` contains a single element, which matches `struct S`. If it does not, then this is a potentially unsafe cast and both `struct T` and `struct S` are made non-updateable. Since our analysis is not context-sensitive, some legal downcasts will be forbidden, for example when a container library is used twice in the program to hold different object types. Fortunately, such context-sensitivity is rarely necessary in the programs we have considered. In the worst case, we inspect the program manually to decide whether a cast is safe or not, and override the analysis results in this case with a `pragma`. We leave to future work the task of more properly inferring polymorphic usage.

5 Dynamic Patches

Patch Generation For each new release we need to generate a dynamic patch, which consists of new and updated functions and global variables, type transformers and state transformers. The Ginseng patch generator generates most of a dynamic patch automatically by comparing the old and new versions of a program to discover the new and modified definitions, and then adds these definitions to the patch file, where unchanged definitions are made `extern`. It also generates type transformers for all changed types by attempting to construct a conversion from the old type into the new type [17]. For example, if a `struct` type had been extended by an extra field, the generator would produce code to copy the common fields and add a default initializer for the added one. This simplistic approach to patch generation is surprisingly effective, requiring few manual adjustments. After the patch is generated and the state and/or type transformers are written, we pass the resulting C file to Ginseng, and the final result is compiled to a shared library so that it can be linked into the running program.

Runtime System To perform an update, the user sends a signal to the running program, which alerts the runtime system. Once the program reaches a safe update point, the runtime system loads the dynamic patch using `dlopen`, checks the validity of the patch and installs it. Ginseng compiles the patch just as it does the initial version of a program, but also introduces initialization code to be run at update-time. The initialization code will effectively "glue" the dynamic patch into the running program by updating the function indirection pointers for all the updated functions, installing the type transformers for the updated types, and running the user-provided state transformer function, if any. Prior to this, it makes sure that the constraints imposed by the updating analysis on the current program point are satisfied by the patch; if not then the update is delayed until the next possible update point.

Our current runtime system has two main limitations. We do not support patch unloading, so old code and data will persist following an update. Fortunately, this memory leak has been minimal in practice—between 21% and 40% after three years' worth of patches for our benchmark applications. Second, dynamic updates are not transactional. If, during an update, an error is encountered, we do not yet have a safe mechanism to abort the update and restore the state to the pre-update one. We plan to address these problems in future work.

6 Experience

We have used Ginseng to dynamically update three open-source programs: the Very Secure FTP daemon (`vsftpd` (`http://vsftpd.beasts.org`), the OpenSSH `sshd` daemon (`http://www.openssh.com`), and the `zebra` routing daemon from the GNU Zebra routing package (`http://www.zebra.org`). We chose these programs because they are long-running, maintain soft state that could be usefully preserved across updates, and are in wide use. For each program we downloaded releases spanning several years and then applied the methodology shown in Figure 1. In particular, we compiled the earliest release to be updateable and started running it. Then we generated dynamic patches for subsequent releases and applied them on-the-fly in release order, while the program was actively performing work (serving files, establishing connections, etc.).

With this process, we identified key application features that make updating the applications easy or hard. We also identified strong points of our approach (that enabled most of the updates to be generated automatically), along with issues that need to be addressed in order to make the updating process easier, more flexible and applicable to a broad category of applications. In the rest of this section, we describe the applications and their evolution history, and the manual effort required to dynamically update them; identify application characteristics and Ginseng features that make updating feasible; and conclude by reviewing factors that enabled us to meet the challenges set forth in Section 2.

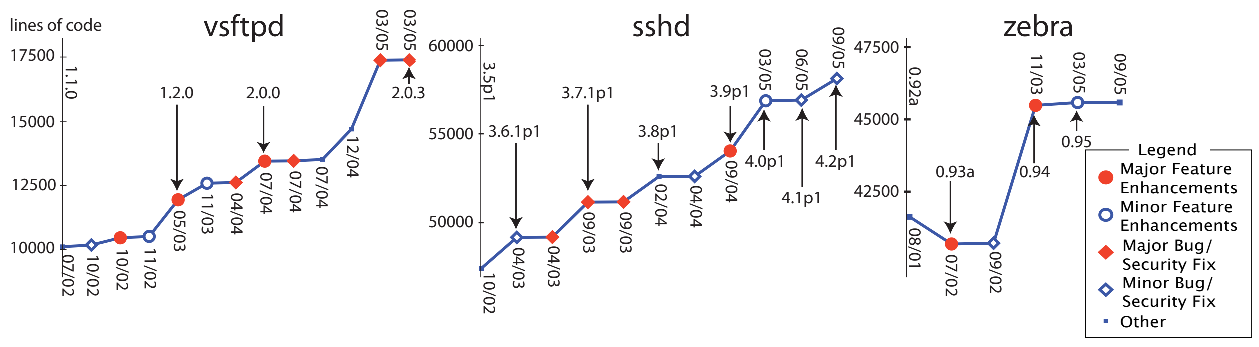

Figure 4: Evolution history of test applications.

Prog.	First release			Last release			Functions				Types			Global variables		
	Ver.	Date	LOC	Ver.	Date	LOC	Add	Del.	Proto changes	Body changes	Add	Del.	Chg.	Add	Del.	Chg.
vstfpd	1.1.0	07/02	10141	2.0.3	03/05	17424	97	21	33	308	12	2	6	72	9	15
sshd	3.5p1	03/02	47424	4.2p1	09/05	58104	131	19	85	752	27	2	19	70	19	29
zebra	0.92a	08/01	41630	0.95a	09/05	45586	134	44	13	321	24	6	4	56	11	52

Table 1: Application update information (all versions).

6.1 Applications

Figure 4 shows the release timeline for each application, along with the nature of individual releases [3] and the code size of each release. We briefly discuss each application first, then describe how the applications changed over a three year period, and finally discuss the manual effort required to dynamically update them.

Vsftpd stands for "Very Secure FTP Daemon" and is now the *de facto* FTP server in major Unix distributions. Vsftpd was first released in 2002. It began to be widely used with version 1.1.0 and is now at version 2.0.3, so for our study, we considered the 13 versions from 1.1.0 through 2.0.3. As can be seen in Figure 4, in the time frame we considered there were 3 major feature enhancements, 3 major bugfixes, 2 minor feature enhancements and 1 minor bugfix.

Sshd is the SSH daemon from the OpenSSH suite, which is the standard open-source release of the widely-used secure shell protocols. We upgraded `sshd` 10 times, corresponding to 11 OpenSSH releases (version 3.5p1 to 4.2p1) over three years.

Zebra GNU Zebra is a TCP/IP routing software package for building dedicated routers that support the RIP, OSPF, and BGP protocols on top of IPv4 or IPv6. It consists of protocol daemons (RIPd, OSPFd, BGPd) and a `zebra` daemon which acts as a mediator between the protocol daemons and the kernel (Figure 5), storing and managing acquired routes. Storing routes in `zebra` allows protocol daemons to be stopped and restarted without discarding and re-learning routes (which can be a time consuming process). We upgraded `zebra` 5 times, corresponding to 6 releases (version 0.92a to 0.95a) over 4 years.

Evolution History Table 1 summarizes the release information and shows some of the ways the programs changed over time. The first two grouped columns describe the first and last release we considered for each program. The last three grouped columns contain the cumulative number of changes that occurred to the software over that span. 'Types' refers to structs, unions and typedefs together. Global variable changes consists of changes to either global variable types or to global variable static initializers. As an example reading of the table, notice that for `vsftpd`, 97 functions were added, 21 were deleted, 33 functions had their prototype changed, and 308 functions had the bodies changed. For `sshd`, 19 types changed; for `zebra`, there were 52 global variable changes.

These statistics make clear that a dynamic software updating system must support changes, additions, and deletions for functions, types and global variables if it is to handle realistic software evolution. Ginseng supports all these

[3] As described at `http://freshmeat.net/`

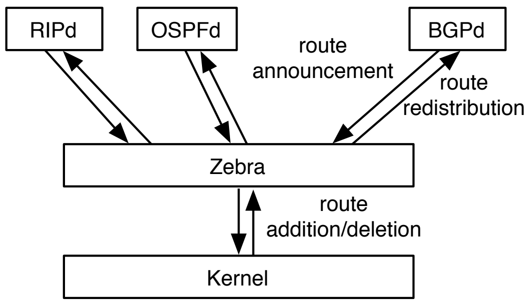

Figure 5: Zebra architecture.

changes, and we have been able to dynamically update the applications from the earliest to the latest versions we considered.

Source Code Changes To safely update these applications with Ginseng required a few small changes to their source code, amounting to around 50 lines of code for `vsftpd` and `sshd` and 40 lines for `zebra`. The changes consisted of introducing named types for some global variables (to support changes in types and static initializers), directives to the compiler (analysis and loop extraction) and in one case (`vsftpd`), instantiating an existential use of `void *`. Another one-line change to `vsftpd` is discussed in the next subsection.

For each new release, we would use the Ginseng patch generator to generate the initial patch, and then verify or complete the auto-generated type transformers and write state transformers (where needed, which was rare). This effort was typically minimal. Table 2 presents the breakdown of patches, across all releases, into manual and auto-generated source code: the first column shows the number of source code lines we had to write for type and state transformers, the second column shows code lines we had to write to cope with changes in global variables' types or static initializers, and the third column shows the amount of code coming out of the patch generator. The code dealing with changes in static initializers for global variables is frequently a mere copy-paste of the variable's static initializer.

6.2 Dynamic Updating Catalysts

In the process of updating the three applications, we discovered four factors that make programs amenable to dynamic updating.

Quiescence. We define a *quiescent point* in the program as one at which there are no partially-completed transactions, and all global state is consistent. Dynamic updates are best applied at such quiescent points, and preferably those that are stable throughout a system's lifetime. Fortunately, each application was structured around an event processing loop, where the end of the loop defines a stable quiescent point: there are no pending function calls, little or no data on the stack, and the global state is consistent. At update time, new versions of the functions are installed and global state is transformed so at the next iteration of the loop will be effectively executing the new program.

For instance, `vsftpd` is structured around two infinite loops: one for accepting new client connections, and one for handling commands in existing connections. Here is the simplified structure:

```
int main() {
  init();
  conn = accept_loop();
  L1:{init_conn(conn);
  handle_conn(conn);}
}
```

```
int accept_loop() {
  L2:while (1) {
    fd = accept();
    if (!fork())
      return fd; }
}
```

```
void handle_conn(fd) {
  L3:while (1) {
    read(cmd,fd);}
}
```

Each time a connection is accepted, the parent forks a new process and returns from the accept loop within the child process. The `main` function then initializes the connection and calls `handle_con` to process user commands. To be able to update the long running loops, and to handle updates following the accept loop in `main`, we used loop extraction (Section 3.4) at each of the three labeled locations so that they could be properly updated. Note that although `L1` is not a loop, by using loop extraction we were able to update code on `main`'s stack (the continuation of `accept_loop()`) without replacing `main` itself.

Application	Source code (LOC)		
	Type/state xform (manual)	Gvar changes (manual*)	Patch gen. auto
vsftpd	162	930	83965
sshd	125	659	248587
zebra	49	244	43173

Table 2: Patch source code breakdown.

A quiescent point is related to, but not identical with a point with empty capability (Section 4); its capability may not necessarily be empty, although it is usually small. On the other hand, an empty capability does not imply quiescence, but rather indicates there are no concrete uses of types beyond the current point.

Functional State Transformation. Our mechanisms for transforming global state (state transformers) and local state (type transformers) assume that we can write a function that transforms old program state into new program state. Unfortunately, sometimes it is not possible to impose the semantics of the new application on the existing state; we encountered two such cases in our test applications. In the upgrade from `sshd` 3.7.1p2 to `sshd` 3.8p1 a new security feature was introduced: the user's Unix password is checked during the authentication phase and if the password has expired, port forwarding will be not be allowed on the SSH connection. However, when upgrading a live connection from version 3.7.1p2 to 3.8p1, the authentication phase has passed already, so the new policy is not enforced for existing connections (though they could be shut down forcibly). For new connections requests coming in after the update, the new check is, of course, performed.

A similar situation arose in going from `vsftpd` 1.1.1 to 1.1.2. The new release introduced per-IP address connection limits by mapping the ID of each connection process with a count related to remote IP address. These counts are increased when a process is forked and decremented in a signal handler when a process dies. Unfortunately, following an update, any current processes will not have been added to the newly introduced map, and so the signal handler will not execute properly. In effect, the new state is not a function of the old state. In this case, the easy remedy is to modify the 1.1.2 signal handler to not decrement the count if the process ID is not known.

When transforming some value, a type transformer can only refer to the old version of the value and global variables, which means that in principle some transformations may be difficult or impossible to carry out. In practice we did not find this to be a problem: for all the 29 type transformers we had to write, the programmer effort was limited to initializing newly added `struct` fields.

Type-safe Programs. As mentioned in Section 4, low-level programming idioms might result in types being marked non-updateable by the analysis. Since having a non-updateable type restricts the range of possible updates, we would like to maximize the number of updateable types, so the solution is to either have a more precise analysis, or inspect specific type uses by hand and override the analysis for that particular type. For the programs we have considered, the techniques presented in Sections 4.2 and 4.3 have significantly increased the precision of the analysis and greatly reduced the need to inspect the program manually. For instance, in `vsftpd`, strings are represented by a `struct mystr` that carries the proper string along with length and the allocated size. The address of the string field is passed to functions, hence revealing `struct mystr`'s representation, but our abstraction violation analysis was able to detect that the aliases were temporary and did not escape the scope of the callee, hence the type was updateable at the conclusion of the call. Polymorphism is employed in all three programs; using the `void *` analysis (Section 4.3) we were able to detect type-safe uses of `void *`, and reduce the number of casts that have to be manually inspected. Inline assembly can compromise type safety as well, and our analysis does not detect type-unsafe uses that might be introduced by assembly code. We only had one such situation in `sshd`, and a manual inspection confirmed the type was used safely. In the end, we manually overrode the analysis only for a handful of types: 0 for `vsftpd`, 1 for `zebra`, and 4 for `sshd`.

Our type wrapping scheme relies on the fact that programs rarely rely on how types are physically laid out in memory; i.e. that they are treated abstractly in this respect. Fortunately, this was a good assumption for these programs. We could not type wrap some "low level" types, e.g., `vsftpd`'s representation of an IP address, since its layout is ultimately fixed by the OS syscall API. On the other hand, this and low-level structures like this one rarely change, since they are tied to external specifications.

Robust Design. We wanted our DSU approach to be general enough to be applied to off-the-shelf software, written without dynamic updates in mind (as was the case with our test applications). However, there are measures developers can take to make applications more update-friendly. Apart from features mentioned above (quiescent points, type safety, and abstract types), we have also found defensive programming and extensive test cases to be helpful in developing and validating the updates. All three programs we looked at were written defensively using `assert` liberally, which facilitated error detection, and helped us spot Ginseng bugs relatively easy. By looking at the assertions in the code, we were able to detect the invariants the programs relied on, and preserve them across updates. `Sshd` comes with a rigorous test suite that provides extensive code coverage, for `zebra` and `vsftpd` we created our own suites to test a broad range of features.

6.3 Summary

We believe we have addressed all the DSU challenges set forth in Section 2. We did not have to change the applications extensively to render them updateable. Patch generation was mostly automatic, and writing the manual parts was easy.

We were able to support a large variety of changes to applications; as can be seen in Table 1 and Figure 4, the applications have changed significantly during the last three years. Once we became familiar with the application structure (e.g., interaction between components, global invariants), writing patches was easy, with all the infrastructure generated automatically; the only manual task was to initialize newly added fields, write state transformers, or make some small code changes.

A combination of factors have helped us address these challenges: (1) programs were amenable to dynamic updating (easily identifiable quiescence points the application, application changes that allowed updates to be written as functions from the old state to the new state, robust application design and moderate use of type-unsafe, low-level code), and (2) Ginseng, especially analysis refinements and support for automation, has made the task of constructing and validating updates easy, even for applications in the range of 50-60 KLOC.

7 Performance

In this section, we evaluate the impact of our approach on updateable software. We analyzed the overhead introduced by DSU by subjecting the instrumented applications to a variety of ’real world’ tests. We considered the following aspects:

1. *Application performance.* We measured the overhead that updateability imposes on an application’s performance by running ’real world’ stress tests. We found that DSU overhead is modest for I/O bound applications, but significant for CPU-bound ones.

2. *Memory footprint.* Type wrapping, extra version checks and dynamic patches result in an increased memory footprint for DSU applications; we found the increase to be negligible for updateable and updated applications, but after stacking multiple patches, the memory footprint increase is detectable.

3. *Service disruption.* We measure the cost of performing an actual update while the application is in use. The update will cause a delay in the application’s processing, while the patch is loaded and applied, and will result in an amortized overhead as data is transformed. In all the updates we performed, even for large patches, we found the update time to be less than 5 ms.

4. *Type wrapping overhead.* In order to measure the impact of type wrapping on CPU-bound applications, we instrumented an application that performs computations on named types exclusively—KissFFT. We found type wrapping to introduce a significant overhead, in terms of both performance and memory footprint.

We also measured the running time of Ginseng to compile our benchmark programs, to measure the overhead of compilation and our analyses.

We conducted our experiments on dual Xeon@2GHz servers with 1GB of RAM, connected by a 100Mbps Fast Ethernet network. The systems ran Fedora Core 3, kernel version 2.6.10. All C code, generated by Ginseng or otherwise, was compiled with `gcc` 3.4.2 at optimization level `-O2`. We have compiled and run the experiments with optimization level `-O3`, but apart form a slight increase in memory footprint (less than 1%), there was no detectable difference in performance. Unless otherwise noted, we report the median of 11 runs.

Application	Connection time (ms)			
	stock	updateable	upd. once	streak
vsftpd	6.71	6.9	7.04	8.4
sshd	47.62	49.26	49.5	62.89

Application	Transfer rate (MB/s)			
	stock	updateable	upd. once	streak
vsftpd	7.95	7.95	7.97	7.98
sshd	7.85	7.84	7.83	7.84

Table 3: Server performance.

7.1 Application Performance

In order to assess the impact of updateability on application performance, we tried different 'real world' stress tests on the updateable applications. For each application, we measure the performance of its most recent version under four configurations. The *stock* configuration is the application compiled normally, without updating. The *updateable* configuration is the application compiled with updating support. The *updated once* configuration is the application after performing one update, whereas the *updated streak* configuration is the application compiled from its oldest version and then dynamically updated multiple times to bring it to the most recent version; this configuration is useful for considering any longer-term effects on performance due to updating.

Vstfpd. We tested `vsftpd` performance with two experiments: connection time and transfer rate. For connection time, we measured the time it took `wget` to request 500 files of size 0, and divided by 500. Since `wget` opens a new connection for each file, and disk transfers are not involved, we get a picture of the overhead DSU imposes on FTP clients. As seen in Table 3, the updateable, updated and streak-updated versions were 3%, 5% and 25% slower than the stock server. With a difference of at most 1.7 ms, we don't believe this to be a a problem for FTP users. We also measured the median transfer rate of a single 600 MB file to a single client. The results are shown in Table 3; the transfer rates of the different configurations are virtually identical.

Sshd. For `sshd` we measured the same indicators as for `vsftpd`, connection time and transfer rate. For the former, we blasted the server with 1000 concurrent requests, and measured the total elapsed time, divided by 1000. (Client-server authentication was based on public key hence no manual intervention was needed.) Each client connection immediately exited after it was established (by running the `exit` command). The measured connection time is shown in Table 3. The updateable, updated and streak-updated versions were 3%, 4% and 32% slower than the stock server. Again, we don't think the 15ms difference is going to be noticed in practice. The CPU-intensive nature of authentication and session key computation accounts for SSH connection time being almost 10 times larger than for FTP. To measure the sustained transfer rate over SSH we used `scp` to copy a 600MB file. As shown in Table 3, the results are similar to the `vsftpd` benchmark—the DSU overhead is undetectable.

Zebra. Since `zebra` is primarily used for route proxying and redistribution, the focus of `zebra` experiments was different than for `vsftpd` and `sshd`. First, we measured the overhead DSU imposes on route addition and deletion by starting each protocol daemon alone with `zebra`, and have it add and delete 100,000 routes. When passing routes through the updatable, updated and streak-updated versions of the `zebra` daemon, the DSU overhead was 4%, 6% and 12%, compared to the stock case (first three clusters in Figure 6). Second, we measured route redistribution performance. We started the RIP daemon, turned on redistribution to OSPF and BGP daemons, made RIP add and delete 100,000 routes, and measured the time it took until the route updates were reflected back into the OSPF and BGP routing tables. Similarly, we timed redistribution of OSPF routes to RIP and BGP daemons. BGP redistribution is not supported by `zebra`. The DSU overhead in the route redistribution case (last two clusters in figure 6) is the same as for the 'no redistribution' case above: 4%, 6% and 12% respectively.

`Zebra` offers a command line interface for remote administration, so as a sanity check only, we measured the connection time for `zebra` as well. We wrote a simple client that connects to the `zebra` daemon, authenticates, executes a simple command ('show version') and the exits. We measured (table 3) a 3% / 3% / 6% increase in connection times for the updatable, updated once and streak-updated `zebra` versions.

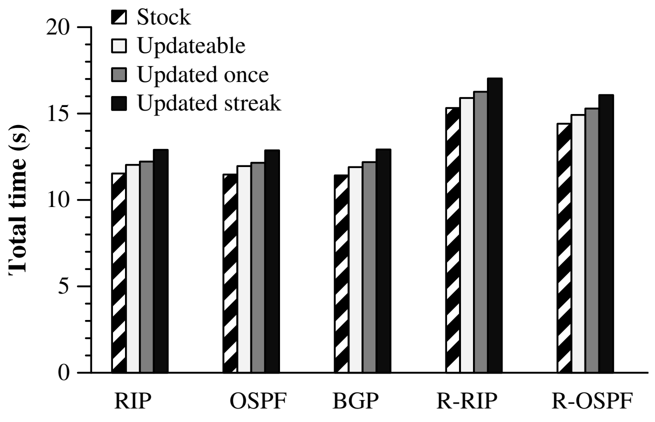

Figure 6: Zebra performance.

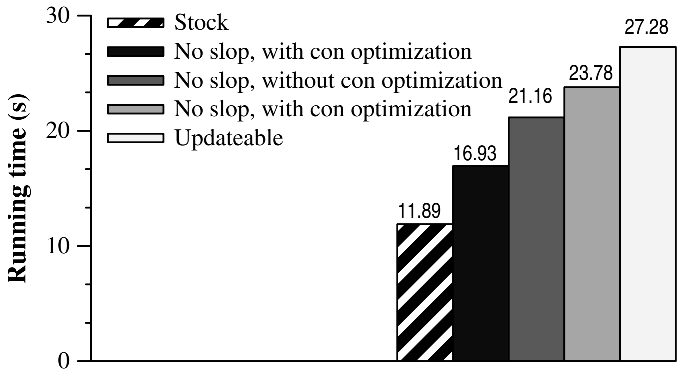

Figure 7: KissFFT: impact of optimizations on running time.

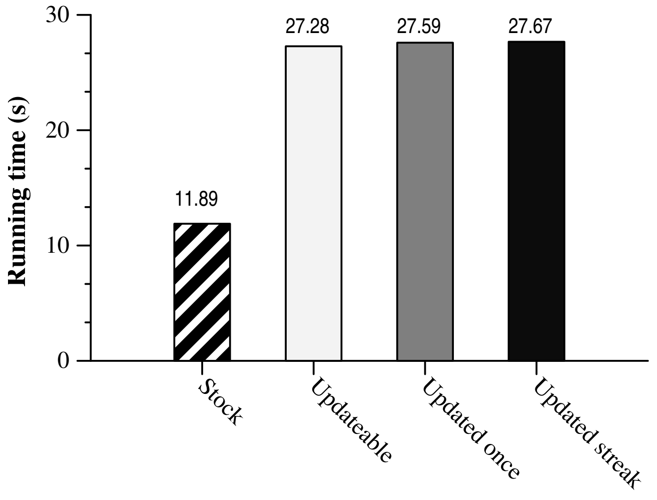

Figure 8: KissFFT: DSU impact on performance.

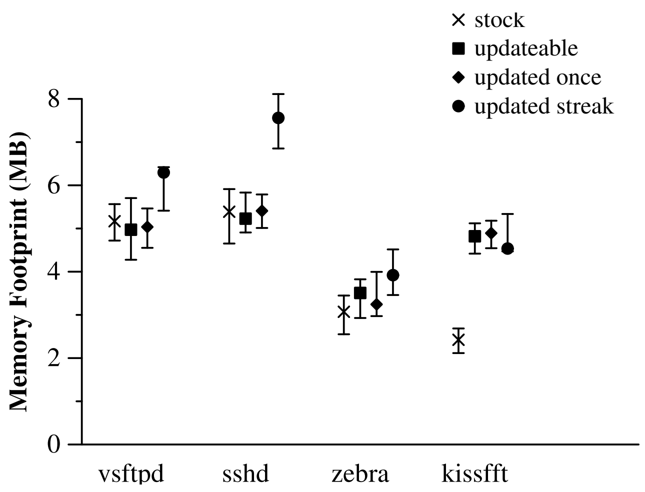

Figure 9: Memory footprints.

KissFFT The overhead of DSU is dwarfed by I/O costs in our experiments. On the one hand, this is good because illustrates that for a relevant class of applications, DSU is not cost-prohibitive. On the other hand, it does not give a sense of the costs of DSU for more compute-bound applications. To get a sense of this, we instrumented KissFFT(`http://sourceforge.net/projects/kissfft`), a Fast Fourier Transform library. Figure 8 shows the total time to perform 100000 Fast Fourier Transforms on 10000 points. The updateable, updated once and updated streak versions were on average 129% slower than the stock version.

We analyzed KissFFT to understand the source of the overhead. The program operates on a large array of complex numbers, and each complex number is represented as a `struct complex`. Therefore, before accessing fields a `__con_complex` has to be performed. Moreover, each complex number will have some slop to accommodate future growth.

Together, these two overheads can make a significant difference, as shown in Figure 7. First, the compiler does not attempt to optimize away redundant `__cons`; that is, KissFFT will perform consecutive `__cons` for data that could not have been updated in between. As shown in the figure, hand-optimizing away redundant cons in the main loop yielded some improvement. Second, the added slop results in poor cache behavior, as far fewer complex numbers in the array would be hot in the cache. The figure shows the effect of setting the slop to 0, effectively just adding the version field to the `struct`. Avoiding redundant `__cons` reduces the DSU penalty to 100%, eliminating the slop reduces the DSU penalty to 78%, and combining the two techniques yields a final DSU overhead of only 42%.

We believe that in the future we'll be able to leverage static analysis in order to avoid introducing redundant `__cons`, and we shall explore different updateable type representations (such as the hybrid solution described in Section 2) for reducing the overhead of the slop.

7.2 Memory Footprint

Type wrapping, function indirection, version checking and loop extraction all consume extra space, so updateable applications have larger memory footprints. Figure 9 reports memory footprints for the four scenarios, with quartiles as error bars. Measurements were made using `pmap` at the conclusion of each throughput benchmark. For the updateable and updated cases, the only significant increase is displayed by KissFFT. The explanation is quite simple: KissFFT uses a large number of `structs` whose size grows by a factor > 2 due to type wrapping. The footprint increases for `vsftpd`, `sshd` and `zebra` are overshadowed by OS variability.

However, for the streak updates, the median footprint increase (relative to the stock version) is 21%, 40% and 27% for `vsftpd`, `sshd` and `zebra` respectively. The larger footprint increase for streak updates is expected, since dynamic patches for three years worth of updates are added into the memory space of the running program, and never unloaded (Section 5).

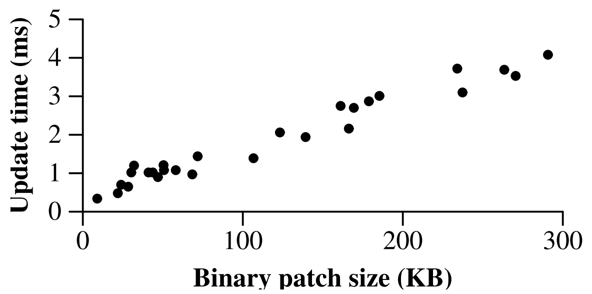

Figure 10: Patch application times.

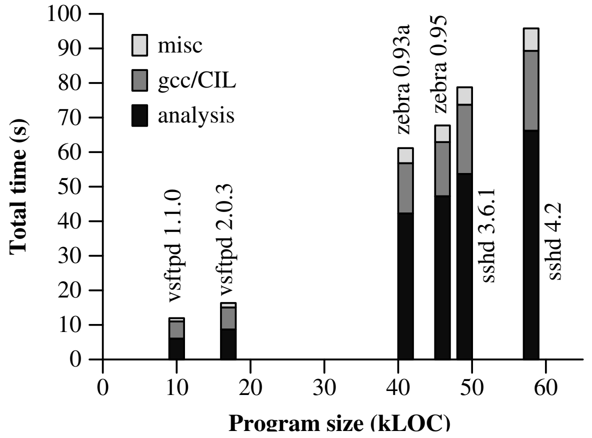

Figure 11: DSU compilation time breakdown for updateable programs.

7.3 Service Disruption

One of the goals of DSU is to avoid service interruption due to the need to apply software patches. By applying these patches on-line, we preserve useful application state, leave connections open, and sustain service. However, the service will still be paused while new patch files are loaded, and service could be degraded somewhat due to the application of type transformers at patch time and thereafter.

Figure 10 illustrates the delay introduced by applying a patch; the delay includes loading the shared object, performing the dynamic linking and running the state transformer (type transformation time was hard to measure, and likely very small, and so is not included). The figure presents measurements for every patch to all of our program versions, and graphs the elapsed time against the size of the patch object files. We can see that patch application time increases linearly with the size of the patch. In terms of service interruption, DSU is minimally intrusive: in all cases, the time to perform an update was under 5 milliseconds.

7.4 Compilation

The time to compile various versions of our benchmarks is shown in Figure 11. The times are divided according to the analysis time (updateability analysis, AVA analysis and constraint solving using Banshee [21]), parsing and compilation time, and remaining tasks. In general, the majority of the overhead is due to the safety analyses, which are whole program, constraint-based analyses. Given that Ginseng is only needed in the final stages of development, i.e., when the application is about to be deployed or when a patch needs to be generated and compiled, this seems reasonable.

8 Related Work

Over the past thirty years, a variety of approaches have been proposed for dynamically updating running software. In this section we compare our approach with a few past systems, focusing on differences in functionality, safety, and updating model.

Updating Functionality A large number of compiler- or library-based systems have been developed for C [13, 16, 9, 2], C++ [18, 20], Java [7, 27, 11, 24], and functional languages like ML [12, 14] and Erlang [3]. Many do not support all of the changes needed to make dynamic updates in practice. For example, updates cannot change type definitions or function prototypes [27, 11, 18, 20, 2], or else only permit such changes for abstract types or encapsulated objects [20, 14]. In many cases, updates to active code (e.g., long-running loops) are disallowed [14, 24, 13, 16, 20], and data stored in local variables may not be transformed [17, 16, 13, 18]. Some approaches are intentionally less full-featured, targeting "fix and continue" development [19, 15] or dynamic instrumentation [9]. On the other hand, Erlang [3] and Boyapati et al. [7] are both quite flexible, and have been used to build and upgrade significant applications.

Many systems employ the notion of type or state transformer, as we do. Boyapati et al. [7] improve on our interface by letting one type transformer look at the *old* representation of an encapsulated object, to allow both the parent and the child to be transformed at once. In our setting, the child will always have to be transformed independent of the parent, which can make writing transformers more complicated or impossible (e.g., if a field was moved from a child object into the parent), though we have not run into this problem as yet. Duggan [12] also proposes lazy dynamic updates to types using type transformers, using *fold*/*unfold* primitives similar to our $con_{\mathrm{T}}/abs_{\mathrm{T}}$. Ours is the first work to explore the implementation of such primitives.

The most similar system is our own prior work on providing dynamic updating in a type-safe C-like language called Popcorn [17]. While that system was fairly flexible, we make three substantial improvements. First, our prior work could not transform data in local variables, could not automatically update function pointers, and had no support for updating long-running loops. We have found all of these features to be important in the server programs, and are part of our current work. Second, while our prior work ensured that all updates were type-safe, it did not ensure they were *representation-consistent* [33], as it permitted multiple versions of a type to coexist in the running program. In particular, when a type definition changed, it required making a *copy* of existing data having the old type, opening the possibility that old code could operate on stale data. Finally, in our prior work we only experimented with a single program (a port of the Flash web server, about 8000 LOC), and all updates to it were crafted by us.

Updating Programs Safely A common theme of prior work is to define "safe states" during a program's execution in which an update may take place. Intuitively, we are interested in the question of whether a change to a system's code, realized dynamically, will properly transform the system to reflect the new code base.

Gupta et al. proved that finding such safe states is, in general, undecidable [16], so any such safety analysis must be conservative. Many of the systems reviewed make no safety guarantees, which can lead to, among other things, run-time type errors [3, 13, 18]. One way to avoid run-time type errors is to sacrifice representation-consistency, as we did in our prior work, mentioned above. Duggan [12] also allows multiple versions of a type to coexist, but avoids the need to make copies of data by requiring a *backward type transformer* to convert data to an older version if it is used by old code; this prevents the problem of stale data. However, it may not always be possible to write backward transformers, since updated types often contain more information than their older versions.

Our current work ensures representation consistency via static analysis; an alternative is to do dynamically. Boyapati et al [7] propose using *transactions* for this purpose. If code in an old object would see an updated object, the current transaction is restarted and old object is itself updated. This basic idea was considered earlier by Bloom and Day [5, 6] in the context of Argus, a system for writing distributed, fault-tolerant applications. We plan to explore the use of transactions in Ginseng in future work.

To avoid the need for rollback, a number of systems aim to ensure safety by relying on a notion of *quiescence*, determined dynamically: only entities not in use by the program may be updated. Dynamic ML [14] supports updating modules M defining *abstract* types t. Since by definition clients of M must use values of type t abstractly, M can be updated to redefine t as long as the old version is inactive and thus not using the old representation. The K42 object-oriented operating system [20, 4] permits updates to objects that are similarly quiescent. It actively achieves this condition by temporarily preventing new threads from calling methods of a to-be-updated object; once existing threads have died, the object is updated and the pending threads may continue. Our safety analysis generalizes these ideas by defining dependency at a finer grain: we check individual uses of types or functions, rather than uses of larger linguistic constructs like objects or modules, which are not directly supported in C.

Updating Models A typical approach to upgrading on-line systems is to use a load-balancer. It redirects requests away from a to-be-updated application until it is idle, at which point it can be halted and replaced with a new version. Such approaches typically employ redundant hardware, which is undesirable in some settings (e.g., upgrading a personal OS). Microvisor [22] employs a virtual-machine monitor (VMM) to follow this basic methodology on a single node. When an application or OS on a server node is to be upgraded, a second OS instance is started concurrently on the same node and upgraded. When the original instance becomes idle, applications are restarted on the new instance and the machine is devirtualized. While Microvisor avoids the need for extra hardware, it shares the same drawbacks as the load-balancing approach: applications must be stateless (so they can be stopped and restarted) or they must be able to save their state under the old version, and then restore the state under the new version. While checkpointing [29, 8] or process migration [30] can be used to stop and restart the same version of an application, it cannot support version changes. DSU handles application state changes naturally. Since all state is visible to an update, it can be changed as necessary to be compatible with the new code. Indeed, one can imagine composing our approach with checkpointing to combine updating with process migration.

9 Conclusions

This paper has presented Ginseng, a system for updating C programs while they run, and shown that it can be used to easily update realistic C programs over long stretches of their lifetimes, with only a modest performance decrease. Our system is arguably the most flexible of its kind, and our novel static analyses make it one of the most safe. Our results suggest that dynamic software updating can be practical for upgrading running systems. We plan to extend our approach to operating systems and multithreaded applications. Ginseng is available for download at `http://www.cs.umd.edu/projects/dsu/`.

References

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proc. PLDI*, 2003.

[2] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *Proc. USENIX Security*, 2005.

[3] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International Ltd., 1996.

[4] A. Baumann, J. Appavoo, D. Da Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing dynamic update in an operating system. In *Proc. USENIX ATC*, 2005.

[5] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT/LCS, March 1983.

[6] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[7] C. Boyapati, B. Liskov, L. Shrira, C-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA*, 2003.

[8] G. Bronevetsky, M. Schulz, P. Szwed, D. Marques, and K. Pingali. Application-level checkpointing for shared memory programs. In *Proc. ASPLOS*, 2004.

[9] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[10] Cristiano Calcagno. Stratified Operational Semantics for Safety and Correctness of The Region Calculus. In *POPL*, 2001.

[11] S. Drossopoulou and S. Eisenbach. Flexible, source level dynamic linking and re-linking. In *Proc. Workshop on Formal Techniques for Java Programs*, 2003.

[12] D. Duggan. Type-based hot swapping of running modules. In *ICFP*, 2001.

[13] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *The Journal of Systems and Software*, 14(2):111–128, 1991.

[14] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, 1997.

[15] A. Goldberg and D. Robson. *Smalltalk 80 - the Language and its Implementation*. Addison-Wesley, Reading, 1989.

[16] D. Gupta. *On-line Software Version Change*. PhD thesis, Indian Institute of Technology, Kanpur, November 1994.

[17] M. W. Hicks. *Dynamic Software Updating*. PhD thesis, The University of Pennsylvania, August 2001.

[18] G. Hjálmtýsson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX ATC*, 1998.

[19] Java platform debugger architecture. This supports class replacement. See `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

[20] The K42 Project. http://www.research.ibm.com/K42/.

[21] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Proc. SAS*, September 2005.

[22] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. ASPLOS*, 2004.

[23] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *POPL*, 1988.

[24] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *Proc. ECOOP*, 2000.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *LNCS*, 2304:213–228, 2002.

[26] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D. A. Patterson, and K. Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Comput.*, 51(2):100–107, 2002.

[27] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of Java software. In *Proc. ICSM*, 2002.

[28] Steve Parker. A simple equation: IT on = Business on. *The IT Journal, Hewlett Packard*, 2001.

[29] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, Computer Science Department, the University of Tennessee, 1997.

[30] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review, SIGOPS*, 22(3):28–40, 1988.

[31] C. Soules, J. Appavoo, K. Hui, D. Da Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. USENIX ATC*, June 2003.

[32] Gareth Stoyle. *A Theory of Dynamic Software Updates*. PhD thesis, Computer Laboratory, University of Cambridge. To appear.

[33] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *Proc. POPL*, 2005.

[34] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[35] Benjamin Zorn. Personal communication, based on experience with Microsoft Windows customers, August 2005.