Dynamic Software Updating*

Michael Hicks
Computer and Information Science Department
University of Pennsylvania
mwh@dsl.cis.upenn.edu

Jonathan T. Moore
Computer and Information Science Department
University of Pennsylvania
jonm@dsl.cis.upenn.edu

Scott Nettles
Electrical and Computer Engineering Department
University of Texas at Austin
nettles@ece.utexas.edu

ABSTRACT

Many important applications must run continuously and without interruption, yet must be changed to fix bugs or upgrade functionality. To date, no existing dynamic updating system has achieved a practical balance between flexibility, correctness, ease-of-use, and low overhead.

We present a new approach that provides type-safe dynamic updating of native code in an extremely flexible manner (functions and types may be updated, and at any time) and permits the use of automated tools to aid the programmer in the updating process. Our system is based around *dynamic patches* made up of proof-carrying code that both contain the updated code and the code needed to transition from the old version to the new. We discuss how patches are generated using a semiautomatic tool, how they are applied using dynamic-linking technology, and how code is compiled to make it updateable.

To concretely illustrate our system, we have implemented a dynamically-updateable web server, FlashEd. We discuss our experience building and maintaining FlashEd. Performance experiments show that updateable FlashEd runs between 2% and 6% slower than a static one.

1. INTRODUCTION

Many computer programs must be "non-stop", that is, run continuously and without interruption. This is especially true of mission critical applications, such as telephone switches, financial transaction processors, airline reservations and air traffic control systems, and a host of others. The increased importance of the Internet and its link with the global economy has broadened needs and has made non-stop service important to a larger range of less sophisticated users who wish to run non-stop services such as e-commerce servers.

On the other hand, companies must be able to upgrade their software to fix bugs, improve performance, or expand functionality. In the simplest case, upgrades and bug fixes require the system to be shut down, updated, and then brought back on-line. This, of course, is not acceptable for non-stop applications, and at best may result in loss of service and revenue, and at worst may compromise safety.

Thus, in general, non-stop systems require the ability to update software without service interruption. Solutions to this problem exist and are widely deployed. A common approach is to provide redundant hardware to support either *hot or cold standbys*. Of course, this approach is expensive and, perhaps worse, adds to the complexity of building applications. Much of the complexity comes from the need for the standby to keep or gain the state maintained by the running application. As an example, Visa makes use of 21 mainframe computers to run its 50 million line transaction processing system. This system is updated as many as 20,000 times per year, but tolerates less than 0.5% downtime [17]. Less sophisticated users do not have Visa's resources, and seek simpler, but no less effective solutions.

We present an approach that enables software updates at runtime that is both cheaper and less complex to use than the above approaches. Our approach, *dynamic software updating*, provides a framework for updating programs as they run and it copes with transitioning state, even when types within the programs change. This framework improves dramatically over existing systems in several areas, including correctness, flexibility, ease-of-use, and low performance overhead.

After stating the goals of our approach in §2, we describe our updating framework in §3 and our implementation of it using TAL [14] in §4. Our experience with a real-world application, a dynamically-updateable web server, *FlashEd*, is presented in §5. We then move on to a more in-depth discussion of existing research and future directions before concluding.

2. GOALS AND APPROACHES

What properties should a dynamic software updating system have? Here, we present several important measures of the practicality of an updating system. We will then briefly argue that existing systems do not satisfy all of the desired properties (although we defer an in-depth discussion of related work to §6). Finally, we describe the major contributions of our system in terms of these same properties:

- **Flexibility.** *Any* part of a running system should be updateable without requiring downtime.
- **Correctness.** A dynamic updating system should promote or guarantee the correctness of updates: mal-

*This work was supported by the NSF under contracts ANI #00-82386 and ANI #98-13875.

formed or otherwise incorrect updates should not cause the running system to crash.

- **Ease-of-use.** Generally speaking, the less complicated the updating process is, the less error-prone it will tend to be. The updating system should therefore be simple to use.

- **Low overhead.** Adding updating infrastructure to a program should impact its performance as little as possible.

2.1 Existing Approaches

Unfortunately, no single existing updating system has *all* of the desired properties. Many systems have limited flexibility, constraining their evolutionary capabilities [8]. For example, dynamic linking is a well-known mechanism, but systems based upon dynamic linking [1, 18] may only *add* new code to a running program, but cannot *replace* existing bindings with new ones. Those systems that do allow updates typically either limit *what* can be updated (*e.g.,* only abstract types [4], whole programs [6], or class instances [10]), *when* the updates can occur (*e.g.,* only when updated code is inactive [4, 12, 19, 6]), or *how* the updates may occur (*e.g.,* functions and values must not change their types [10], or changes to module and class signatures are restricted [12, 4]). These limitations leave open the possibility that a software update may be needed yet cannot be accomplished without downtime.

In many cases, it is difficult to know that the application of an update will not cause a crash in itself. Some systems, for example, break type safety [21, 10, 19, 6] or have only dynamic checking [2], or require potentially error-prone hand-generation of complex patch files [11, 4, 12, 19, 6, 2]. Others rely on uncommon source languages or properties [11, 2] and hence are not broadly applicable. Finally, some systems rely on having interpreted code [12], which rules out their usefulness for various high performance applications (*e.g.*, web servers, transaction systems).

2.2 Our Framework

Our framework, *dynamic software updating*, avoids the need for extra equipment and the high development costs of hot and cold standbys. Furthermore, unlike previous systems, it has all four of the above desired properties, in some cases exceeding the current state-of-the-art.

Flexibility. Our system permits changes to programs at a function granularity, and we permit the types of data and functions to be updated. Updates are permitted to occur at any time, even while the code being updated is active. Our implementation is built with compiler and library support for a safe, C-like language, on top of standard dynamic linking facilities, meaning it is applicable to other similar languages.

Correctness. In our system, dynamic patches consist of *proof-carrying code* [14, 15]. As a result, a patch cannot crash the system or perform many illegal actions since it can be proven to respect important safety properties, including type safety. Our use of standard dynamic linking facilities also means we have not expanded the trusted computing base (TCB), which is the trusted part of the system upon which the security of the entire system rests.

Ease-of-use. Construction of patches is largely automated, with the software development process remaining unchanged. When a new software version is completed, a tool compares the old and new versions of the source files to develop patches that reflect the differences. Although total automation is undecidable, our tool can nonetheless generate useful patch code for a majority of cases, leaving placeholders for the programmer in the other (infrequent) cases.

```
static int num = 0;
int f(int a, int b) {
  num++;
  return a + b;
}
```

Figure 1: A file f

new version f':

```
static int num = 0;
int f(int a, int b) {
  num++;
  return a * b;
}
```

state transformer S

```
void S () {
  f'::num = f::num;
}
```

Figure 2: Dynamic patch for f: (f', S)

Low Overhead. Our system permits dynamic updating of native code, giving obvious performance benefits as compared to interpreted systems like Java. Furthermore, in practice, our updating infrastructure does not place an undue amount of overhead on an application.

In short, our approach provides type-safe dynamic updating of native code in an extremely flexible manner and permits the use of automated tools to aid the programmer in the updating process. No previous system achieves all of these goals together.

3. FRAMEWORK

In this section, we discuss our general framework; details about our implementation of a specific instance of this framework are presented in §4. We assume an imperative source language, using C-like code in the examples.

3.1 Dynamic Patches

Central to our approach is the idea of a *dynamic patch*; namely, one that is applied to a running program. Dynamic patches differ from static patches, such as those created and applied using the Unix programs `diff` and `patch`, because they must deal with the state of the running program. We can abstractly define a dynamic patch of some file f as the pair (f', S), where f' is the new code and S is an optional *state transformer function*, used to convert the existing state to a form usable by the new code.

For example, consider the file f shown in Figure 1. The function `f` increments `num` to track the number of times it was called and returns the sum of its two arguments. Suppose we modify `f` to return the product of its arguments, producing f'. The dynamic patch that converts f to f' is shown in Figure 2. The state transformer function S is trivial: it copies the existing value of `num` in the old version f to the `num` variable in the new version f'. In general, arbitrary transformations are possible.

Because patches apply to individual files, rather than the whole program, our definition is not yet complete. In partic-

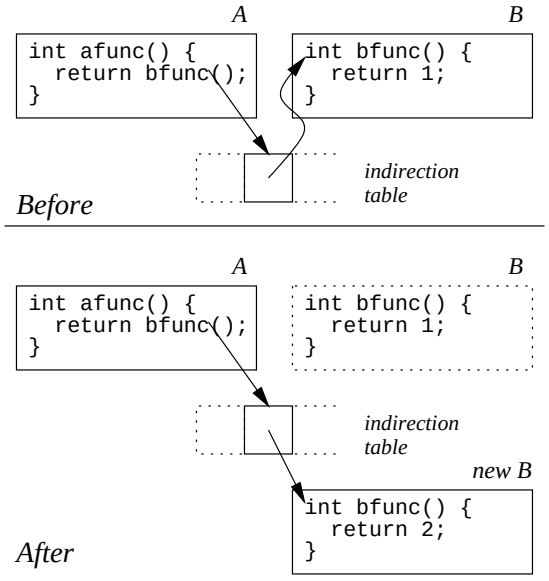

Figure 3: Updating by reference indirection

ular, what happens when new code in f' redefines functions in f to have a *different type*? The problem is that existing callers of `f` will still use the old type, meaning the patch has introduced a type error in the program. One way to prevent this is to simultaneously apply patches to correct the callers. More generally, we can extend the notion of a patch to optionally include *stub functions* to be interposed between old callers and new definitions to get the types right.

Stubs are only useful for functions that change type; there is no analogous construct for data. Thus, if a patch changes the type of some global variable, then all the code that references that data must be simultaneously changed, or else there will be two versions of the same global variable. The simplest case is when the global variable is not exported from the file (*i.e.* it is `static`), since only the functions in the local file itself must be changed.

3.2 Enabling Dynamic Patches

> *"Any problem in Computer Science can be solved with another level of indirection." —attributed to David Wheeler in Butler Lampson's 1992 ACM Turing Award speech*

Here, we consider what mechanisms our running programs will need to support dynamic patches.

3.2.1 Code and Data Updates

Applying a patch requires that references to existing function calls and data be redirected to the stubs and new definitions in the patch. There are essentially two ways to do this: either by *rewriting* or by *indirection*.

Rewriting allows the code to be compiled normally. At update-time the running code is rewritten to refer directly to appropriate parts of the patch. The advantage of rewriting is that it allows normal compilation and incurs no runtime cost except at update time. Instead of rewriting, our approach uses indirection, as shown in the example in Figure 3. Here we wish to perform an update to change `bfunc`, which is referred to by `afunc`. Before the patch `afunc` refers to the original `bfunc` through an indirection table, much like the one used to support dynamic linking. Applying the patch updates the indirection table to point to the `bfunc` defined in the patch.

The main advantage of the indirection approach is that it is simple to implement, but at the cost of a small performance penalty. However, we feel that this penalty is justified for two reasons. First, we have measured it to be negligible in practice—see §4. Second, we note that widely-used dynamic linking approaches, most notably ELF [21], also require an extra level of indirection for external references. In fact, these indirections can be exploited to enable dynamic updating for these systems, as described by [20] for a slightly different context.

3.2.2 Updating Type Definitions

If we wish to preserve type-safety, we need a way to upgrade the *type definitions* as understood by the type-checker used by the dynamic linker. Again, there are basically two approaches we could take: *replacement* or *renaming*. With replacement, applying the patch *replaces* the existing type definition in the typechecking context with a new one. Newly loaded code is checked against the new definition, implying that to preserve consistency we must also convert any existing instances of the old type definition (whether in the heap, stack, or static data area) to the new one. Furthermore, any code that makes use of old type elsewhere in the program must itself be replaced. One exception is in the case that the type is abstract; then only the ADT must be replaced.

In contrast, we maintain a fixed notion of a type definition, and instead rely on the compiler to define a new type that *logically* replaces the old one, syntactically *renaming* occurrences of the old name with the new one. When the patch is applied existing instances of the old type are left as they are; the state transformer function and/or the stub functions in the patch can be used to convert old instances at update-time or later if needed. The typechecking context retains its definition of the old type and adds a new one for the new type.

The benefit of this approach is that no additional runtime support is needed to replace old type instances; again we can rely on standard dynamic linking support. The notable drawback of this approach is that as time goes on, the number of type definitions maintained by the typechecker could grow to be very large. It also requires a standard method for renaming type definitions so that disconnected developers do not choose clashing names. This problem is simply dealt with by taking the MD5 hash of the definition.

3.3 Building Updateable Systems

Now that we understand the mechanisms for building a system that can have dynamic patches applied to it, two key methodological questions remain. The first is how the patches are generated. The second is how to structure our system so that patches can be correctly applied, particularly with respect to the timing of patch application.

3.3.1 Patch Construction Methodology

It should be easy for programmers to generate correct patches. Furthermore, if possible, generating patches should not make the normal process of code development substantially more difficult.

Our approach to generating patches is simple. First, the

programmer develops and tests a new version of the code, exactly as if he was going to statically compile and deploy it. Next, our system automatically generates as much of the patch file as possible by comparing the source of the old code to that of the new code. Finally, the programmer fills in the parts of the state transformer and stub functions that could not be automatically generated.

A key benefit of this approach is that software development is separated from patch development. This is possible because our notion of patch (and our implementation of it) allows essentially arbitrary changes to the running program. In many other systems, patches are limited to certain forms, and so software development is similarly limited. For example, in Dynamic C++ classes [10], changes are limited to instance methods and data; static methods and data cannot evolve. As a result, the process of generating patches is tied to development, with the newest version of the software having artifacts of the old version, such as useless fields in structures or additional copies of static data.

3.3.2 Automatic Patch Generation

A novel aspect of our approach is the (mostly) automatic generation of patch files. This feature was originally born out of convenience: it is very tedious to write state translation and stub functions by hand. It has also proven invaluable in minimizing human error: it is less likely that a necessary state translation or stub function will be accidentally left out. As it turns out, a very simple syntactic comparison of files, informed by type information, can do a good job of identifying most changes.

The job of the patch generator is twofold: identify changes to functions and data, and when possible, generate appropriate stub functions and state transformers. The identification algorithm is simple. First, both the old and new version of the file to patch are parsed and type-checked. Then, for each definition in the new file, the corresponding definition is looked up by name in the old file. In the case of type definitions (*i.e.* `struct` or `union` declarations), the bodies of the definition are compared and differences are noted. In the case of value declarations, the bodies are also compared syntactically, taking into account the differences in type definitions; in particular, the syntax of a function may remain the same from the old to the new version, but the function has actually changed if a type definition mentioned in the body has changed.

After the identification has completed, the state translation code is generated. For all global variables that remain unchanged, an assignment statement is created from the old to the new versions, like the one for `num` in Figure 2. For those global variables that have changed type, appropriate code is generated automatically, when possible. For example, in the webserver we often change the type definition `httpd_conn`, which contains information about a pending connection. All connections are stored in a global array of `httpd_conn`s. In this case, the generator automatically inserts a loop that copies from the old to new array, calling a type conversion function for each element, which is also generated automatically (to the extent possible), as explained below.

The patch generator also generates default stubs for functions that have changed type. Two basic modes are possible. In the simplest mode, the generator merely inserts a statement that raises an exception. This is useful when all patches for the running program are to be applied simultaneously. In this case no stub functions should ever be invoked, so the exception signals an unexpected error. The second mode is to automatically generate a call to the new version of the function, first translating the arguments appropriately, like the case depicted in Figure ??. Because we have, to this point, only applied all patches simultaneously, we have not yet implemented this mode, although it will be straightforward.

During the identification phase, the patch generator keeps track of any type definitions that have changed, and generates new names for these types. The new name is determined by taking the MD5 hash of the pretty-printed type definition (for uniformity). This allows development of patches by multiple programmers without the worry of choosing incompatible type names.

Finally, *type conversion functions* are constructed to the extent possible for data conversion from old to new versions of a type, and vice versa. These are used by the state transformation and stub code, as mentioned above. For `struct` types, each field with an unchanged type is copied; each field that is added is given a default value; and each field that has changed type is translated. In the case that a translation is not possible, a placeholder is left for the programmer to fill in the appropriate value. Currently we support translation between like types (*i.e.*, `int` and `float`), and `struct` and `union` types (by calling the appropriate type conversion function).

3.3.3 When to Apply Patches

A critical component of assuring patch correctness is the *timing* of an update. In particular, it is possible for a well-formed update to be applied at a bad time, resulting in incorrect state. For example, consider the file f and its patch, shown in Figures 1 and 2, respectively. Here the patch state translation function S copies the current value of `num` to the new version. The new code then uses this new version of `num`. If this patch is applied while `f` is *inactive* (that is, `f` is not currently running, and not on the stack somewhere) then everything will be fine. However, if (the old version of) `f` begins execution just before the patch is applied, it will increment the *old* version of `num` *after* it has been copied by S. The result is the new version of `num` will not reflect the call of `f`.

Unfortunately, Gupta has shown that the problem of correct timing is, in general, undecidable [6]. Thus, in existing systems, programmers must identify correct timing conditions for a given patch, a task which typically must be done by hand [11, 3] or with very limited automated support [6]. Furthermore, automatically enforcing these conditions requires special runtime support [11] or restrictions to updating only inactive code [4, 12, 19], which still does not necessarily guarantee that race conditions of the above sort will not occur.

Instead, we observe that the problem of timing can be greatly simplified by requiring the program to be coded from the outset so that updates are only permitted at well-understood times. This transfers the timing enforcement issue from run-time to compile-time: rather than assuming, as past approaches do, that a program will not be aware that it is updateable, and thus updates may conceptually occur at any time, we instead require the program to be coded to perform its own updating. Furthermore, not only can we

'eyeball' the code to determine an appropriate spot, we can use the techniques of previous authors mentioned above to determine one. The difference is that this spot is codified at *software construction time*, as opposed to specified and enforced at runtime.

As a result, we avoid the implementation complexity of update timing enforcement, without losing the benefits of correctness. The cost is that the system must be constructed appropriately from the outset. However, given that the clientele of dynamic updating systems already recognize the need for updates, this is perfectly reasonable. Our own experience, and that of other updating systems, such as Erlang [2], indicate that this burden is not great, especially compared to the complexity of hot and cold standbys.

4. IMPLEMENTATION

We have implemented our framework to target Typed Assembly Language (TAL) [14], using the source language Popcorn [13], a safe subset of C. Beginning with with a brief introduction to TAL and Popcorn, this section presents the details of our implementation. First, we describe how we implement dynamic updating by reference indirection. Then we explain how we define and compile patch files.

4.1 TAL and Popcorn

TAL is a cousin of proof-carrying code [15], a framework in which native machine is coupled with annotations such that a safety proof can be checked. A type-correct TAL program is memory safe (*i.e.* no pointer forging), control-flow safe (*i.e.* no jumping to arbitrary memory locations), and stack-safe (*i.e.* no modifying of non-local stack frames) among other desirable safety properties. TAL has been implemented for the Intel IA32 instruction set; this implementation, called TALx86 [13], includes a TAL verifier and a prototype compiler from a safe-C language, called Popcorn, to TAL.

4.2 Dynamic Updating

In previous work, we added a type-safe dynamic linker to TALx86 [9]. Our current work extends that work to provide dynamic updating for Popcorn programs. We briefly describe the existing dynamic linker, and follow with the changes we made to support dynamic updating.

The TAL dynamic linker consists of two parts, a *trusted* part (written in C and OCaml), and a *untrusted* part, written in Popcorn. The novelty of this linker is that almost all of its functionality is written in Popcorn, and can thus be proven type-safe. In particular, all of the functionality related to linking and symbol management occurs within the untrusted part; only loading and verification occurs in the trusted computing base. As a result, the linker is more trustworthy.

All program files (whether dynamically or statically linked) are compiled so that their external references are indirected through a local table called the *global offset table* (GOT) in the style of ELF dynamic linking [21]. At link time, the entries in this table are resolved with the exported definitions of the running program. These definitions are tracked by the dynamic linker within a global *dynamic symbol table*. In ELF, both the GOTs and the dynamic symbol table are a trusted part of the object file header, but in our system, they are written in Popcorn. In particular, the GOT for each loadable file is constructed automatically via a source-to-source translation, and the dynamic symbol table is within the untrusted part of the dynamic linker. As a result, the indirection facility and the the process of linking can be checked for type-safety.

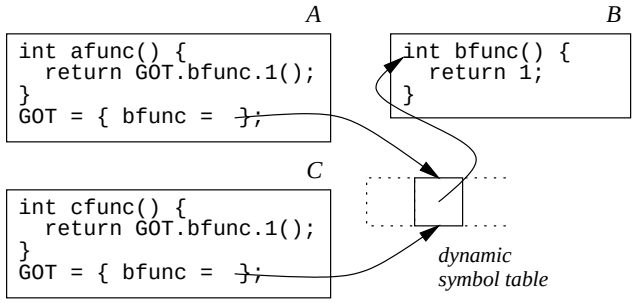

Figure 4: Indirection via the Dynamic Symbol Table

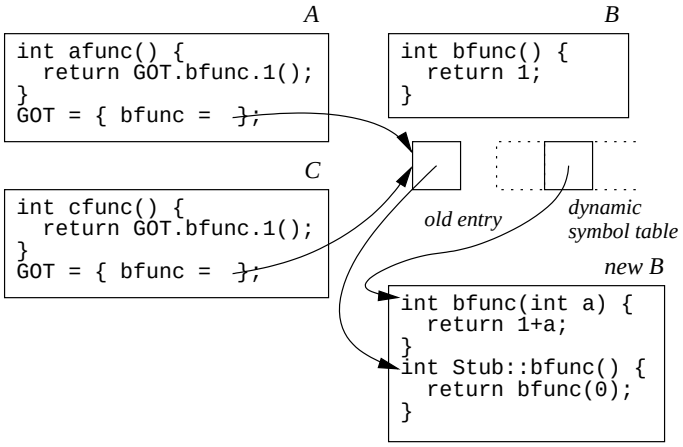

Figure 5: Following a dynamic update of B

To support dynamic updating, we modified this basic approach as follows: first, rather than filling each GOT entry with its corresponding definition, we instead fill it with that definition's entry in the dynamic symbol table. This is illustrated in Figure 4. Both `afunc` and `cfunc` indicate that the `bfunc` field from the GOT structure should be extracted (`GOT.bfunc`), dereferenced (`GOT.bfunc.1`), and finally called (`GOT.bfunc.1()`).

When a file is updated, the old entry in the dynamic symbol table is replaced with the new one, redirecting the callers. In the case that the new function changed type and there is a stub function, the old entry is *removed* from the table, and redirected to point at the stub function. A new entry is added to the table that points to the new function. This is shown in Figure 5. Function `b` has been updated to take an integer argument, and a stub function is included to redirect the old callers. The old module B is now unreachable, and can be garbage-collected.[1]

In addition to this change, we required some other alterations to make everything work:

[1] Unless it was actually being executed at the time of update; in this case, it will be unreachable as soon as the PC leaves the module.

- *Rebinding.* We can map symbols in the program to different names in the dynamic symbol table. This allows us to replace function symbols with stubs that do not have the exact same name.

- *Customized linking order.* This allows us to look up existing table entries before they are overwritten.

- *Exporting* `static` *variables.* This allows state transformers have access to all global state. To avoid name clashes, we experimented with different ways of modifying local variable names. We settled on prepending local variables with *filename*`::Local::`.

Some additional points are important. First, the GOT only stores references to externally-defined variables; references to locally-defined variables are direct. This means that only a subset of all function calls and data references made by a program will have to pay the indirection penalty; we present measurements of how significant this is in §5. At the same time, this approach essentially requires updates to occur on the granularity of files, rather than individual data or procedures. Otherwise, the callers of an updated procedure from the original file would still call the old version. We believe this tradeoff will significantly reduce the cost of indirections, while having only a small impact on flexibility.

Second, our approach differs slightly from the abstract approach outlined in §3.2; we use two indirections per external reference, rather than one. Using only one indirection would require the dynamic symbol table to track, for each symbol, all the addresses of GOT entries that point to its value. Then, when the symbol is updated in the dynamic symbol table, the new value would be propagated to all of the local GOTs to reflect the change. We do not use this technique simply because Popcorn's `&` operator only works on global variables, due to limitations of the TAL verifier, so there is no way to take the address of a GOT field to be stored in the dynamic symbol table. However, we expect the TAL implementation team to relax this restriction, at which time we will optimize our implementation.

4.3 Patches

Our implementation of dynamic patches closely follows the abstract description of §3.1. The contents of a patch are described by a *patch description file* containing four parts: the implementation filename, the interface code filename, the shared type definitions, and the type definitions to rename. The first two fields describe the patch: its implementation in the first file, and the state transformer and stub functions in the second file. The final two fields are for type namespace bookkeeping. The shared type definitions are those types that the new file has in common with the old, while the changed definitions are in the renaming list, along with a new name to use for each. The compiler uses this information to syntactically replace occurrences of the old name with the new one.

As introduced in the state translation function of Figure 2, we need a way to refer to different versions of a variable within the interface code file. For a variable x, we may wish to differentiate between the *old* version of x, the *new* version of x, or the *stub function* for x. This is achieved by prepending the variable references in the interface code file with `New::`, `Old::`, and `Stub::` respectively. With no prefix, the reference defaults to the version available before the patch was applied; this turns out to simplify how we compile patch files.

The patch file is prepared for compilation by first converting it into a normal Popcorn file, which is then compiled for dynamic updating. The translation works as follows. First, all definitions in the implementation file whose variables are in the sharing list are made into `extern`s, which will resolve to the old version's definitions at link time. Second, all of the defined variables (non-`extern`) in the implementation file are prefixed with `New::`. Third, the interface code file and the implementation file are concatenated together. Finally, all the mappings from the renaming list are applied to the file's type names. The resulting file is then compiled to be loadable and updateable, as described in the previous subsection.

5. THE FLASHED WEBSERVER

To demonstrate our system, as well as to further inform its design and implementation, we developed a dynamically-updateable webserver, based on the Flash webserver [16]. The original Flash consists of roughly 12,000 lines of C code and is one of the highest performance web servers available today. We call our server **FlashEd** for *Ed*itable *Flash*. We use FlashEd to illustrate three points about our system: how to construct an updateable application, how we constructed and tested patches in practice, and the performance implications of our approach.

5.1 Building an Updateable Application

Flash's structure is well-suited to our requirements for update validity. It is constructed around an event loop (in a file separate from that of `main`) that does three things. First, it calls `select` to check for activity on client connections and the connection listen socket. Second, it processes any client activity. Finally, it accepts any new connections. Note that this kind of event loop is common in most server applications.

Only two changes were needed to Flash to support dynamic updating. First, we added a *maintenance command interface*. A separate application connects to the webserver and sends a textual command with the files to dynamically load. After the `select` completes, a pending maintenance command is processed and the specified dynamic patches are applied. Upon completion, the event loop exits back to `main`, which then restarts the loop (thus reflecting any change to the file containing the loop) and continues processing. Existing state is preserved between loop invocations.

The second change was in how errors were handled. Flash contained many places where errors were detected and program execution aborted by calling `exit`. Such aborts are not acceptable in a non-stop program. We changed these cases to throw an exception instead. The event loop catches any unexpected exceptions, prints diagnostics, shuts down existing connections, and restarts. If an exception is thrown from a file (module) that maintains state, that state is also reset. Thus the program can continue service until it can be repaired online, albeit with the loss of some information and connections.

5.1.1 Patching

To gain experience evolving a program using our system, we constructed FlashEd *incrementally*. Our initial implementation lacked some of the C version's features (such as

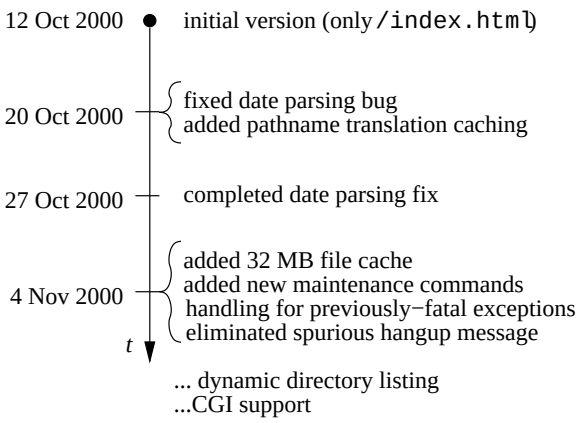

Figure 6: Timeline of *FlashEd* updates

CGI and directory listings) and performance enhancements (such as pathname translation caching and file caching). We added these features, one at a time, following the process outlined in §3.3.1. Version two adds pathname translation caching; version three adds file caching; and version four, still in progress, will add directory listings and CGI.

The changes between each version are non-trivial. A number of types change, including the `structs` defining HTTP connections, file information, and pathname translations; and significant functionality is added and modified. Of the original eighteen sources files, eleven of them changed from version one to two, and nine changed from two to three, resulting in sixteen and fourteen patch files, respectively. The additional five patch files in each case were as a result of changes to type definitions; no code in these files was changed, but the structure of data changed, requiring recompilation.

Patch construction was relatively easy. The automatic generator did most of the work, identifying which types and code had changed and creating default patch and interface code files. For the version two patch, we modified four patch files, adding or modifying a total of about 30 lines of code; for version three we modified five files and added or changed 65 lines of code. These changes included initialization statements for variables normally initialized by `main` and some state translation code.

5.2 Experience

To simulate a production environment, we are running a public server and attempting to never shut it down, making all changes on-line. A brief chronology for FlashEd is shown in Figure 6. We started version one at `http://flashed.cis.upenn.edu` on October 12, 2000, to host the FlashEd homepage. We applied patches for version two on October 20 and for version three on November 4. All patches were tested offline on a separate server under various conditions, and when we were convinced they were correct, we applied them to the on-line server. Even so, we found a mistake in the first patch—a flag had not been properly set—and applied a fix on October 27. In addition, we applied roughly five small patches for debugging purposes, such as to print out the current symbol table.

Running the server has revealed which aspects of the system work well and which do not. For instance, we learned soon after we deployed the server that our version of the TAL verifier is buggy—it only checks a subset of all of the basic blocks in loaded files. Since the verifier is part of the trusted computing base, it cannot be updated. Ultimately we must shut down the system and recompile it with the new version. To accommodate these kinds of changes, we could allow certain trusted code to be loaded without benefit of verification.

We also made a human error when compiling the server: we forgot to enable the exporting of `static` variables when compiling the library code. This problem became apparent when we attempted to dynamically update the dynamic updating library. The library was not properly removing old entries from the dynamic symbol table, and so we wanted to patch the library to fix the problem, as well as clean up the existing symbol table. However, since the symbol table is declared `static`, it was not available for use by the patch. As a result, any update to the library is effectively precluded since the state cannot be properly transferred.

On the whole, however, the system has been extremely easy to use. It has been particularly effective to be able to load code to print out diagnostic information. For example, on a number of occasions we loaded code that would print out the dynamic symbol table (by calling an existing function in the updating library) to make sure that symbol names referenced in our patches, particularly the ones chosen for static variables, matched the ones present in the table. We also loaded code to print out the state of the file and translation caches, to make sure that things were working.

Having the verifier to check patches as they are being loaded has been essential. For example, we tried to apply some patch files that were incorrectly generated; the implementation file path mentioned in the patch description file was for an incorrect version. As a result, some of the type definitions were incorrect, and this fact was caught by the verifier. Once we applied a patch whose state translation function failed to account for null instances; the updating library caught the `NullPointer` exception and rolled back the changes made to the symbol table. Using an unsafe language, such as C, would have resulted in our non-stop system stopping with a core dump.

5.3 Performance Analysis

Adding dynamic-updating imposes a number of costs on the system. At update-time, each patch must be verified and linked. However, this one-time cost is easily amortized across the lifetime of the program. As for run-time costs, each reference to global data outside of the referring file must go through two extra indirections. In this section, we present the results of some experiments that measure these costs.

Our experimental cluster is made up of four dual-300 MHz Pentium-II's with split first level caches for instruction and data, each of which is 16 KB, 4-way set associative, write-back, and with pseudo LRU replacement. The second level 4-way set associative cache is a unified 512 KB with 32-byte cache lines and operates at 150 MHz. These machines receive a rating of 11.7 on SPECint95 and have 256 MBs of EDO memory. Each machine is connected to a single Fast Ethernet (100 Mb/s), switched by a 3Com SuperStack 3000. We run RedHat Linux 6.1, which uses Linux kernel version

20 clients

Webserver	static	updateable	updated
FlashEd v1	50.1	48.3	*n/a*
FlashEd v2	49.9	48.5	49.2
FlashEd v3	52.2	51.2	51.7

100 clients

Webserver	static	updateable	updated
FlashEd v1	70.6	66.5	*n/a*
FlashEd v2	72.7	69.2	70.6
FlashEd v3	84.5	82.5	83.1

Table 1: FlashEd Throughput (Mbits/sec)

2.2.12.

5.3.1 FlashEd performance

To measure server performance, we used WebStone (version 2.5), a freely available webserver benchmarking system [22]. WebStone allows a specified number of client processes to be forked on multiple machines, sending HTTP `GET` requests to the server. Each client receives an identical *filelist* containing a list of files to request, and a corresponding probability for each file. Each request is determined pseudo-randomly, as preferenced by this probability. After a specified time, all clients are halted, and relevant data from each is collated. While other performance metrics are potentially interesting, here we focus on *server throughput*, which is the total bytes served by the webserver divided by the total time.

We ran our tests using the recommended configuration. We used the standard filelist, which contains files of sizes 500B, 5KB, 50KB, 500KB, and 5 MB, where the most preferred size is 5 KB, followed closely by 500B. We ran for configurations of 20 and 100 clients, evenly distributed across three machines; the server itself ran on the fourth machine. Each run was for 10 minutes. We measured 11 runs and the data presented here are means. In all cases the standard deviation was 1% or less than the mean and the distributions were not skewed.

To understand the cost of updating, we compared statically-compiled and updateable versions of all three versions of FlashEd. In addition, we compared the difference between a version that was statically compiled with updating enabled, to the version that was actually patched on-line. We suspected that the latter case might have worse performance due to a larger memory footprint or poorer cache locality. In particular, it will retain the original version of the code in the text segment along with any new code, which is loaded into the data segment. Table 1 shows the results of our measurements, reported in Mbits/sec.

The difference in throughput between the first and second columns is due to adding indirections. This difference ranges from 2% to 6% and tends to be greater for version 1 and larger numbers of clients. The difference between the second and third column is the additional overhead created by actually updating the running system. Surprisingly, the updated code is consistently faster than the statically linked code. These differences are negligible, and certainly within the margin of error of our experiments, but we are currently attempting to find a reason nonetheless.

As a point of reference, we measured the fully-optimized, C version of Flash. For 20 clients it achieves an average of 56.8 Mbit/sec and for 100 clients 70.2 Mbit/sec. We expected Flash to consistently outperform FlashEd, so we are surprised that for version 3 of FlashEd in particular, the reverse is true. However, although FlashEd is a faithful port of Flash, it is not clear what exactly to draw from this comparison beyond the fact that this suggests that TAL, and PCC in general, is a viable platform for medium-performance applications.

5.3.2 Microbenchmarks

To gain a more detailed understanding of the cost of the indirections, we measured null function calls using the Pentium cycle counter. A call to a local function (not requiring any indirection) was 36 cycles, while a call to an external function (requiring two indirections) was 46 cycles, a difference of 10 cycles (28%). In a more optimized implementation having only a single indirection, the difference is 7 cycles (19%).

The indirection penalty occurs only when references are to definitions not in the current file. To discover the relevant fraction of references, we modified the Popcorn compiler to insert counters for global references, whether to data or functions, differentiating between references to external and static variables. We then re-built the three versions of FlashEd and ran our macrobenchmark suite on each of them for times of one, three, and ten minutes, with 20 clients.

For versions one, two, and three of the server, the dynamic reference count was 70%, 71%, and 62%, respectively. Combining this information with the directly measured, per-reference overhead, we arrive at average reference times of between 42 and 43 cycles, or between 17% and 19% slower than without updating (the optimized implementation would be between 10% and 14% slower). Our macrobenchmark results show that, in practice, this is a very loose upper bound.

Another important cost is the time to verify and link the loaded patch files. To patch version one to two, the total time for the sixteen files was roughly 17.7 seconds, on average about 1.1 seconds per file. For the fourteen files patching version two to three, the total time was roughly 18.3 seconds, or about 1.3 seconds per file. According to [5], verification is generally linear in the size of the files being verified. Overall, these times are disappointing. However, they do not outweigh the value of verification for two reasons. First, verification times could very well be improved. For example, proof-carrying code [15] has demonstrated small verification times, albeit with a different type system. Second, verification could be performed in parallel with normal service. After verification completes, only linking remains, and this cost is negligible; in the two sets of patches above, linking accounted for 130 ms and 125 ms of the total time, respectively.

6. DISCUSSION

To conclude, we discuss related work, place our current work into a broader context, and consider future work. We organize the discussion around our four major criteria for evaluating updating systems: flexibility, correctness, ease-of-use, and low overhead. A more complete discussion of related work may be found in [7].

6.1 Flexibility

At one extreme of the flexibility axis are systems that use

dynamic linking alone to support updating [1, 18]. These solutions are only adequate when the programmer anticipates the kinds of updates that may be made ahead of time and structures the program to accommodate them. This is because dynamic linking only allows new code to be plugged into existing interfaces. The lack of flexibility of this kind of system is a direct inspiration for our current work [8].

Some systems are more powerful than dynamic linking, but do not allow arbitrary changes. For example, Dynamic ML [4] permits the replacement of ML modules whose types are *abstract*, as long as the new module's signature is not incompatible with the old one: existing elements cannot be removed and their types must be unchanged. A similar restriction is placed on the Dynamic Virtual Machine [12], a Java VM with updating ability, and Dynamic C++ classes [10].

At the other extreme of the flexibility axis are systems that, like ours, allow nearly arbitrary changes to programs at runtime, thus supporting true program evolution [11, 3, 6, 2, 12, 10]. DYMOS [11] (DYnamic MOdification System) is perhaps the most flexible existing system; it supports multi-threaded programs, allows changes to occur on a per-function or per-module basis, and also provides for infinite loops to be updated. Like our system, some of these systems mitigate complexity and add flexibility by permitting updates to occur while old code is still active. A gradual transition from old to new code occurs at well-defined points, such as at procedure calls [2, 11, 3], or during object creation [10].

We believe our system represents a good compromise between the two extremes: the generality of our dynamic patches allow us to achieve most of the flexibility of the most general solutions, and programmer control of patch application gives good flexibility in timing updating. However, there are some important flexibility limitations we would like to address, as informed by our experience with FlashEd.

Unchecked updates. The most obvious limitation is that because we only allow updating to occur using proof-carrying code, it is impossible to update the trusted computing base. Although we expect changes at that level to be rare, they also can be critical, as we saw with the need to update the verifier so that it verifies all parts of a patch. Although there are some technical difficulties (not to mention correctness and safety issues), we could relax this limit by providing a lower-level interface to the dynamic loader, circumventing the PCC-only one we use in general (as described in [9]).

Function pointers. We also have difficulty dealing with function pointers. In our approach, updating is enabled by adding an extra indirection at each reference to an external function *name* in the code. However, no provision is made for data that points to external *values*. For instance, a function pointer to some function `f` would still point to the old version if `f` were updated. While we can work around this problem during state transformation, we have experimented with solving this problem more generally by having the compiler alter its translation to dereference function pointers late, rather than early. In particular, in the current implementation, the actual *address* of `f` is stored, while in the experimental one, a *pointer to the address* of `f` is stored instead. This allows the dereferencing of the function pointer to occur just before it is called, obtaining the newest version. There are a number of problems with this approach that we are trying to work out; doing so will be crucial for our methodology to apply to functional languages.

Namespace Security. Throughout this paper we have assumed a model in which one or more trusted implementors may change the software and update its running code. However, in some circumstances, we would like to allow updates from untrusted sources. For example, we might build a webserver in which users load *servlets* to perform some customized operation on their behalf. User code should not have access to core components of the webserver, or to other servlets, implying the need to control the symbol namespace used during linking, based on some security criteria.

This is easy to do for the value namespace in our system, thanks to how we have constructed the TAL dynamic linker [9]. It should be simple to modify the untrusted part of the dynamic linker to support policy-based access to value symbols. In fact, such a change could even be realized dynamically! However, doing the same for types, whose definitions are stored in the trusted part of the dynamic linker (the verifier), is less straightforward. One possibility is to parameterize the loading primitive with a first-class type environment. This environment will be assumed to be a subset of the environment as understood by the verifier, a fact that can be checked. Then the untrusted part of the linker can be programmed with policies to restrict the typing environment at appropriate times.

Updating abstract types. In Popcorn, structures and unions can be declared `abstract`, meaning that only the code in the local file may see the type's implementation; this is enforced by the TAL verifier. As a result, no dynamically-linked file will be able to see the implementation of an abstract type. In general, this behavior is desirable, but it also prevents us from loading new code to "update" (by replacement) the implementation of the abstract type.

It is possible that the proposal we outlined above for namespace security can apply to abstract types as well. In particular, the verifier can maintain the least restrictive type environment (that allows breaking the abstraction), while the untrusted linker code will pass in a more restrictive environment for all those cases except ones in which the type's implementation is to be updated.

6.2 Correctness

Dynamic linking provides a significant advantage with respect to correctness over the more general updating system we have proposed, simply because bindings are stable: once bound, a reference never changes. Previous work has leveraged this fact to try to build support for evolving systems that only use dynamic linking. For example, Appel [1] describes an approach in which the old and new version of code can run concurrently in separate threads, with the old version phasing out after it completes its work. Similarly, Peterson *et al.* [18] describe an application-specific means of stopping a program, updating its code, and then invoking the new version with the old version's state. Both of these approaches suffer the problem that they are more difficult to use and less flexible.

However, as we explained in §3.3.3, allowing code to change arbitrarily can result in incorrect behavior if timing is not considered. While we believe that our approach of requiring the system to be constructed so that a correct update point

is determined, much work remains for determining *where* such safe points lie. In particular, things get more complicated with multithreading. Previous work [6, 11, 3] can serve as a starting point for this investigation.

Correctness is greatly strengthened by verifying important safety properties of loaded code, including *type safety*. This is a key benefit to our approach, and to the DVM [12], which makes use of Java bytecode verification. Other systems benefit from the use of type-safe source languages, like SML [1, 4], Haskell [18] and Modula [11], but must trust the compiler; we need only trust the verifier. Erlang is dynamically typed, so runtime type errors are possible. Most other approaches are for C [3, 6] and C++ [10], which lacks the benefit of type-safety.

6.3 Ease-of-Use

Dynamic linking is generally easy to use and is well integrated into standard programming environments. Also due to its widespread support in current languages and systems, it is also quite portable. In contrast, the more flexible systems are quite hard to use. In all of the existing systems, patches must be constructed by hand: the programmer must identify parts of the system that have changed and reflect these in the file to load. In many cases, the limitations of patch files hamper the normal development process.

Ease-of-use is one of the areas that our system makes the greatest contributions. Our basic methodology, in which programs are developed normally, and dynamic patches update the old version to the new, limits disruption of normal work flow. In particular, the semi-automatic generation of patches greatly increases the ease-of-use of our system, automating the most tedious parts of patch generation, while letting the programmer control the more subtle aspects that are not amenable to automation.

6.4 Low Overhead

While both Gupta's system [6], and Dynamic ML [4] essentially implement updating by rewriting (as defined in §3.2), most systems implement dynamic updating by indirection, either as we do [2, 12, 10], or in slightly more clever ways [11, 3]. Dynamic linking may, as in the case of ELF [21], or may not impose an indirection as well, affecting those systems based on it [18, 1]. As we have demonstrated here, however, this extra indirection does not translate to high overhead in practice. Furthermore, because our approach is based on native code, it lacks the overhead of interpretation, *e.g.*, as in the DVM [12].

7. CONCLUSIONS

We have presented a system for dynamic software updating built on type-safe dynamic linking of native code. Our framework provides significant advances in balancing the tradeoffs of flexibility, correctness, ease-of-use, and low overheads, as borne out by our experience with our dynamically updateable webserver, FlashEd.

8. REFERENCES

[1] A. Appel. Hot-sliding in ML, December 1994. Unpublished manuscript.

[2] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

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

[4] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without Dynamic Types. Technical Report ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, The University of Edinburgh, December 1997.

[5] D. Grossman and G. Morrisett. Scalable certification for Typed Assembly Language. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

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

[7] M. Hicks. Dynamic software updating. Technical report, Department of Computer and Information Science, University of Pennsylvania, October 1999. Thesis proposal.

[8] M. Hicks and S. Nettles. Active networking means evolution (or enhanced extensibility required). In *Proceedings of the Second International Working Conference on Active Networks*, October 2000.

[9] M. Hicks, S. Weirich, and K. Crary. Safe and flexible dynamic linking of native code. In *Preliminary Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, Technical Report CMU-CS-00-161. Carnegie Mellon University, September 2000.

[10] G. Hjálmtýsson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[11] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, April 1983.

[12] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the Fourteenth European Conference on Object-Oriented Programming*, June 2000.

[13] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[15] G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, Jan. 1997.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable webserver. In *Proceedings of the USENIX Annual Technical Conference*, pages 106–119, Monterey, 1999.

[17] D. Pescovitz. Monsters in a box. *Wired*, 8(12):341–347, 2000.

[18] J. Peterson, P. Hudak, and G. S. Ling. Principled dynamic code improvement. Technical Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997.

[19] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, pages 53–65, March 1993.

[20] A. Serra, N. Navarro, and T. Cortes. DITools: Application-level support for dynamic extension and flexible composition. In *Proceedings of the USENIX Annual Technical Conference*, March 2000.

[21] Tool Interface Standards Committee. Executable and Linking Format (ELF) specification, May 1995.

[22] Mindcraft—webstone benchmark information. `http://www.mindcraft.com/webstone`.