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. No prior general-purpose methodology for dynamic updating achieves a practical balance between flexibility, robustness, low overhead, and ease of use.

We present a new approach for C-like languages that provides type-safe dynamic updating of native code in an extremely flexible manner (code, data, and types may be updated, at programmer-determined times) and permits the use of automated tools to aid the programmer in the updating process. Our system is based on *dynamic patches* that both contain the updated code and the code needed to transition from the old version to the new. A novel aspect of our patches is that they consist of *verifiable native code* (*e.g.* Proof-Carrying Code [17] or Typed Assembly Language [16]), which is native code accompanied by annotations that allow on-line verification of the code's safety. We discuss how patches are generated mostly automatically, 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 for FlashEd, the overhead due to updating is typically less than 1%.

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 financial transaction processors, telephone switches, 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 made non-stop service important to a larger range of less sophisticated users who wish to run e-commerce servers.

On the other hand, companies must be able to upgrade their software to fix bugs, improve performance, and 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; at best, it will 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 application-specific software support in conjunction with redundant hardware (already present to support fault tolerance) to enable so-called *hot standbys*. For example, Visa makes use of 21 mainframe computers to run its 50 million line transaction processing system; it is able to selectively take machines down and upgrade them by preserving relevant state in the on-line computers. This system is updated as many as 20,000 times per year, but tolerates less than 0.5% downtime [20]. Of course, Visa's approach is expensive and, perhaps worse, adds to the complexity of building applications. Much of the complexity comes from the need for the standby machine(s) to keep or gain the state maintained by the running application.

Less sophisticated users do not have Visa's resources, and seek simpler, more general, but no less effective solutions. In particular, while redundant hardware may often be present to support fault tolerance, we would prefer not to *require* it for updating, since it adds cost and complexity. By using a simpler, general-purpose approach, we can support systems that do not typically require extra hardware, like communications components (*e.g.* routers, firewalls, NAT translators, *etc.*), simple Internet servers, monitoring systems, and others. Furthermore, there are many non-redundant systems that do not necessarily *require* non-stop service but would certainly benefit from it. For example, rather than having to reboot a desktop computer each time its operating system is upgraded, we would prefer to realize the updates dynamically.

We present a general-purpose framework for updating programs as they run, called *dynamic software updating*, that is flexible, robust, easy to use, and efficient. Our approach is both cheaper and less complex than typical application-specific approaches, and as we shall argue, improves significantly over existing general-purpose systems.

After stating the goals of our approach in §2, we describe our updating framework in §3 and our implementation of it

*This work was supported by the NSF under contracts ANI #00-82386, ANI #98-13875, and ANI #0081360.

using Typed Assembly Language [16] in §4. Our experience with a real-world application, a dynamically-updateable web server, *FlashEd*, is described in §5; its performance is presented in §6. We then move on to a more in-depth discussion of existing research and future directions before concluding. This work summarizes the first author's thesis [10]; readers seeking more explanation and analysis should look there.

2. GOALS AND APPROACH

What properties define an effective dynamic updating framework? To evaluate general-purpose dynamic updating systems, we establish four evaluation criteria:

- **Flexibility.** *Any* part of a running system should be updateable without requiring downtime.
- **Robustness.** A system should minimize the risk of error and crash due to an update, using automated means to promote update correctness.
- **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 easy to use.
- **Low overhead.** Making a program updateable should impact its performance as little as possible.

In this section, we first argue that existing systems do not satisfy all of these criteria (we defer an in-depth discussion of related work to §7). Next, we describe the key aspects of our approach, and explain how they combine to successfully meet these requirements.

2.1 Existing Approaches

Unfortunately, no existing general-purpose updating system meets *all* of the desired criteria. Many systems have limited flexibility, constraining their evolutionary capabilities; for example, dynamic linking is a well-known mechanism, but while systems based upon dynamic linking [2, 21] may *add* new code to a running program, they cannot *replace* existing bindings with new ones. Those systems that do allow replacement typically either limit *what* can be updated (*e.g.,* only abstract types [7], whole programs [9], or class instances [12]), *when* the updates can occur (*e.g.,* only when updated code is inactive [7, 14, 6, 9]), or *how* the updates may occur (*e.g.,* functions and values must not change their types [12], or changes to module and class signatures are restricted [14, 7]). These limitations leave open the possibility that a software update may be needed yet cannot be accomplished without downtime.

In many cases, there are few safeguards to ensure update correctness. Some systems, for example, break type safety [22, 12, 6, 9, 5] or have only dynamic checking [3], or require potentially error-prone hand-generation of complex patch files [13, 7, 14, 6, 9, 3, 5]. Others rely on uncommon source languages or properties [13, 4, 3] and hence are not broadly applicable. Finally, some systems impose a high overhead, either due to implementation complexities [13, 5], or due to a reliance on interpreted code [14].

2.2 Our Framework

Our framework, *dynamic software updating*, avoids the extra equipment and added complexity of typical application-specific approaches, and unlike previous general-purpose systems, it meets all four of the evaluation criteria through a novel combination of new and existing technology.

Flexibility. Our system permits changes to programs at the granularity of individual definitions, be they functions, types, or data. Furthermore, we allow these definitions to change in arbitrary ways; most notably, functions and data may change type, and named types may change definition. The system does not restrict when updates may be performed, even allowing active code to be updated. Our approach uses an imperative, C-like language, and should thus be widely usable.

Robustness. In our system, dynamic patches consist of *verifiable native code*, in particular Typed Assembly Language (TAL) [16]. As a result, a patch cannot crash the system or perform many incorrect actions since it can be proven to respect important safety properties, including type safety; ours is the first dynamic updating system to use verifiable native code. Our implementation builds on top of basic dynamic linking, keeping the implementation simple and robust.

Ease of use. Construction of patches is largely automated and clearly separated from the typical development process. 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. No previous system both cleanly separates patch development from software development and provides automated support for patch construction.

Low Overhead. Our system imposes only a modest runtime overhead, largely inherited from dynamic linking. Because we use TAL, programs and patches consist of native code, giving obvious performance benefits as compared to interpreted systems like Java.

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. As a result, ours is the first dynamic updating system to satisfy all of the evaluation criteria.

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 are *dynamic patches*; namely, ones that are 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*

```
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)

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 are applied to individual files, rather than whole programs, there is a problem in applying a single patch if exported code or data *changes type*: existing referers of changed items will access them at the old (now incorrect) type. In general, this problem can be 'corrected' by simultaneously applying patches to correct the callers. In most situations, it would not make sense to do otherwise; in transitioning from one version of a program to another, it only makes sense to patch all of the files that changed. However, for greater flexibility, 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. There is no analogous construct for data, so if a patch changes the type of some global variable, then all the code that references that variable *must* be simultaneously changed.

3.2 Enabling Dynamic Patches

For reasons of flexibility and simplicity, we build dynamic patch application on top of dynamic linking. In essence, a patch is dynamically linked into the running program, and then the running program is transitioned from the old code to the new code. In this section, we consider possible mechanisms for transitioning the program to use dynamically linked patches. We arrive at our basic methodology by considering possible choices in light of our evaluation criteria, particularly flexibility, efficiency, and ease of use. We first consider updates to code and data, and then updates to type definitions.

3.2.1 Code and Data Updates

Once a patch has been dynamically linked into the program, existing function calls and data must be redirected to the stubs and new definitions in the patch. There are essentially two ways to do this: either by *code relinking* or by *reference indirection*. When using code relinking, the rest of the program is relinked after loading a patch; as a result, all references to the old definitions will be redirected to refer to the new ones. By contrast, reference indirection requires modules to be compiled so that references to other modules are indirected through a global indirection table. An update then consists of loading a patch and altering appropriate entries in the table to point to the patch.

With relinking, the process of updating is *active*: the dynamic linker must go through the entirety of the program and 'fix up' any existing code to point to the new code. With reference indirection, updating is *passive*: the existing code is compiled to *notice* changes. As a result, the linker does not need to keep track of the existing code and simply makes changes to the table, but at the cost of an extra indirection to access definitions through the table. In both cases it is the responsibility of the state transformer function to find references to old definitions that are stored in the program's data. For example, if the program defines a table of function pointers, the state transformer must redirect each pointer in the table to its new version.

We have chosen to use code relinking because it has two main benefits: it avoids extra indirection, reducing overhead, and it is simple to implement, enhancing robustness. In particular, we implement code relinking by reusing the code in the dynamic linker (described in §4.1). The only apparent burden is the need to keep track of the existing code to be able to relink it; but we must do this already, since the dynamic linker resolves external references in loaded code against all the existing code.

Ultimately, we could take a hybrid approach in which some elements are compiled to notice updates, and others must be relinked. One possibility that we have explored is to compile pointerful data (notably function pointers) to have an extra indirection, but require code references to be relinked. This would ease the requirement that the state transformer translate pointer data. We touch on this idea further in §7.

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 the old type elsewhere in the program must itself be replaced (unless the type is abstract; then only the code that implements the type must be replaced).

The alternative approach is type renaming. Instead of allowing type definitions to be replaced, we maintain a fixed notion of a type definition, and rely on the compiler to define a new type that *logically* replaces the old one by syntactically *renaming* occurrences of the old name with the new one. Renaming is similar to the idea of α-conversion in scoped programming languages, in which a type definition can override a definition of the same name in a surrounding scope; the overriding type is renamed to avoid the clash. The consequence of this approach is that 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.

There are advantages to both approaches. Type replacement, in general, is quite flexible and easy to use: it maintains the identity of a type within the program but lets its definition change. The system updates the values of the changed type (perhaps using user-provided code), so long as the programmer has updated all of the modules that use that type. However, because the program has no notion of the old and new versions of the type, the system must ensure that it can (logically) convert all of the old types invisibly. This restriction prevents an update to a type while code in the program is using values of that type [7]. In contrast, type renaming only allows the loading of new types to logically replace existing ones, placing more burden on the programmer to convert values from the old to new type in either the state transformer or stub functions. However, renaming provides more freedom in timing updates, since the program is 'aware' of both versions.

Implementing type renaming is quite simple, requiring no additional runtime support. To be practical it does require a standard method for renaming type definitions at compile-time so that different developers do not choose clashing or inconsistent names, which would result in program errors. This problem can be solved by taking a cryptographic hash (*e.g.* using MD5 [18]) of the type's definition to arrive at a consistent name. In contrast, type replacement requires a way to find all existing instances of a given type, and a way to change them from the old version to the new. Furthermore, to ensure that type updates do not occur when code that uses them is active requires heavyweight mechanisms to track when modules are in use [6, 13, 9].

We favor the simpler type renaming approach over the more complex, though easier to use, type replacement approach. Type renaming is more likely to be correctly implemented because it is simple, and is more portable, relying on facilities available in type-safe dynamic linkers. Renaming also provides more flexibility as to when and how values of changed type will be converted. Other approaches [14] have cited runtime type dispatch operators (*e.g.* `instanceof` in Java) as a reason for performing type replacement, but we believe more study is needed to bring to light the problems of type renaming in such a context. In our experience, renaming types at compile-time, and having multiple notions of a type in the program, has not been problematic; we present some of our experience in this regard in §5.1.1.

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

Constructing correct software is already difficult, so having to write dynamic patches only compounds the difficulty. A key goal of our approach has been to reduce the added burden as much as possible. In particular, we wish to simplify the process of constructing patches, and we wish cleanly to separate software development from patch development.

Our approach to generating patches is straightforward. 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, and all of these changes are encapsulated in the patch file. In many other systems, patches are limited to certain forms, and so software development is similarly limited. For example, in Dynamic C++ classes [12], 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 transformation and stub functions by hand. It has also proven invaluable in minimizing human error: it is less likely that a necessary state transformation 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, 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 transformation 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 FlashEd, our updateable webserver (described in §5), 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. Because we have, to this point, only applied all patches simultaneously, we have not yet implemented this mode, although it should 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 by taking the MD5 hash of the pretty-printed type definition. 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. Values of `union` type are deconstructed and then reconstructed at the new type, translating any fields that have changed. 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 transformation 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) 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 [9]. Thus, in existing systems, programmers must identify correct timing conditions for a given patch, a task which typically must be done by hand [13, 6] or with very limited automated support [9]. Furthermore, dynamically enforcing these conditions requires special runtime support [13] or restrictions to updating only inactive code [7, 14, 6], 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 [3], indicate that this burden is not great, especially compared to the complexity of application-specific approaches that use hot and cold standbys.

4. IMPLEMENTATION

We have implemented our framework to target Typed Assembly Language (TAL) [16]. Both TAL and its cousin, proof-carrying code [17], belong to a framework we call *verifiable native code*, in which native machine code is coupled with annotations such that the code is provably *safe*. A well-formed 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 properties. TAL has been implemented for the Intel IA32 instruction set; this implementation, called TALx86 [15], includes a TAL verifier and a prototype compiler from a safe-C language, called Popcorn, to TAL.

This section presents the details of our implementation, including how we implement dynamic updating by code relinking, and how we define patch files and compile them to TAL.

4.1 Dynamic Updating

In previous work, we added a type-safe dynamic linker to TALx86 [11]. 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.

At the core of the TAL dynamic linker is a simple primitive, load, that loads and verifies TAL modules. The remainder of the linker's functionality, which includes linking and symbol management, is written in Popcorn, and can thus be proven type-safe, adding to the implementation's robustness.

Dynamically loadable files 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 [22]. At load-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*. This 'table' consists of a linked list of hashtables, one per module, that maps symbol names to their addresses. In ELF, both the GOT and the dynamic symbol table are encoded as 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 generated and maintained by symbol management part of the dynamic linker. As a result, the indirection facility and the the process of linking can be checked for type-safety.

To support dynamic updating, we alter this scheme only slightly. Say we are loading a patch for some program module A.

1. *All* files, whether statically- or dynamically-linked, are compiled to have a GOT, and external references are indirected through that GOT.

2. When the patch for A is loaded, a new hashtable is created to be stored in the dynamic symbol table. Once the patch has been linked with the running program, the patch's state transformer copies the old state from A, transforming it as necessary. If an error occurs during linking (*e.g* a symbol is looked up at the wrong type) or state transformation (some exception is raised), then we *roll back* to the old version of A. This can be done by simply throwing out the new hashtable, since the old code and the old hashtable has not been modified. Once state transformation is complete, the existing code in the program is relinked; this includes the present version of A in case that code is still active. The result is that the GOT's of each of the existing files will have their entries redirected to the new code's symbols. Finally, the old A's hashtable is essentially removed from the dynamic symbol table (see below). When applying multiple patches simultaneously, we require more than one linking pass, since patches may contain mutually-recursive references, but the gist is the same.

To properly support these operations, we modified our dynamic linker to support the following features:

- *Exporting* `static` *variables.* This allows state transformers have access to all global state. To avoid name clashes between files, we prepend local variables with *filename*`::Local::`.

- *Customized linking order.* This allows us to look up existing table entries before they are overwritten; this is important for state transformer functions, which may refer to both old and new versions of a given variable.

- *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.

- *Secondary lookups.* After a patch is loaded, say for module A, the old version of A needs to be relinked in case it is still active. In this case, if a lookup during the relinking finds a requested symbol at the wrong type, it secondarily looks for the old version of that symbol in an older hashtable. This circumstance will only occur when a symbol changes type and does not, or cannot in the case of data, define a stub function. Because the old A is going to shortly be outmoded by the new A, we allow the code to use the old version of the symbol. In contrast, when relinking the rest of the program (*i.e.* everything but the old version of A), we do not allow secondary lookups, effectively enforcing that current code always refers to the most current symbols.

- *Weak pointers.* Once relinking is complete, we would like to remove any old hashtables from the dynamic symbol table to make the old code unreachable, and thus garbage-collectible. However, doing so is strictly correct because there is no guarantee that this old code will not be active at the next update, and thus need to be relinked. An effective compromise is to keep the old hashtables linked into the dynamic symbol table with *weak pointers*. Weak pointers do not keep data from being garbage collected when not reachable by some non-weak pointer elsewhere in the program, and thus code can be collected when it is no longer needed.

  Unfortunately, TAL does not currently support weak pointers, but adding them would be straightforward. To simulate the weak pointer implementation, for purposes of understanding the performance of updated programs, we remove the old tables following an update, but ensure via program construction that the removed code will not be active at the next update.

4.2 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 transformation 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 compiled by translating it into a normal Popcorn file, and then using the normal Popcorn compiler. 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 above.

To ver	changed files	changed types	Δ source LOC	total patches	interface LOC auto	interface LOC by hand
0.2	11	3	433	16	1324	48
0.3	9	2	813	14	1261	99
0.4	7	1	1557	12	1214	99

Table 1: Summary of changes to versions 0.2 through 0.4 of FlashEd

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 [19]. Flash consists of roughly 12,000 lines of C code and has performance competitive with popular servers, like Apache [1]. We constructed our version, called *FlashEd* (for *Ed*itable *Flash*), by recoding Flash in Popcorn while preserving its essential structure and coding techniques. In this section, we use FlashEd as a case study to explain three aspects of our system: how to construct an updateable application, how to construct and test patches in practice, and how dynamic updateability affects application performance; we look at the first two of these points in this section, and discuss performance in the next section.

5.1 Building an Updateable Application

Flash's structure is quite amenable to ensuring patches are well-timed. 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. This kind of event loop is common in 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 and re-enters the loop (thus reflecting any change to the file containing the loop) and continues processing. Relevant state is preserved between loop invocations.

The second change was to how errors were handled. Flash contains many places where `exit` is called upon the discovery of illegal conditions. Such aborts are not acceptable in a non-stop program, so we changed these cases to throw an exception instead. When the event loop catches any unexpected exceptions, it prints diagnostics, shuts down existing connections, and restarts. If an exception is thrown from a module that maintains state, that state is also reset. Thus the program can continue service until it can be repaired, 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 (version 0.1) lacked some of the Flash's features (such as dynamic 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 0.2 adds pathname translation caching; version 0.3 adds file caching; and version 0.4 adds dynamic directory listings.

Information about the changes between versions, including the patches that resulted, is summarized in Table 1. Columns two to four of the table show the changes to the source code made from the previous version, including the number of changed or added source files (not including header files), the number of changed type definitions, and the number of changed or added lines of code. The last three columns describe the patches, including the total number of patches generated (not including the type conversion file), the total lines of generated code for the patch interface code files, and those lines that were added or changed by hand.

There are two things to notice in the table. First, the number of patches generated exceeds the number of changed source files; this is because certain type definitions changed, such that functions in those files that refer to those types also effectively changed. Second, the number of lines of interface code automatically generated far exceeds the amount modified or added by hand. This is not to say that the process of modifying the automatically-generated files was simple (it was not in some cases), only that a large portion of the total work, much of it tedious, could be done automatically. For example, many of the generated lines include `extern` statements that refer to the old and new versions of changed definitions; these would have had to be placed by hand otherwise. Most importantly, using the patch generator guaranteed that the patches were *complete*—all of the changes were identified automatically, even though some changes needed to be addressed by the programmer.

The alterations to the generated files usually had one of three forms. First, we had to write code in the state transformer function to translate pointerful data. For example, sometimes various *connection handler functions* changed, so we had to translate references to those handlers in the global handler array to point to the new version. Second, we had to occasionally fill in placeholders in the generated type conversion functions, particularly for function pointers (like the per-connection timeout function) and newly-defined types (like a `struct` added to manage cached files). Finally, we had to add code to the state transformer to initialize new functionality; this code already exists in (and was copied from) `main`, but because the new functionality is added dynamically, the code must run in the state transformer.

5.2 Experience

To simulate a production environment, we have been 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 3. We started version 0.1 at `http://flashed.cis.upenn.edu` on October 12, 2000, to host the FlashEd homepage. We applied patches for version 0.2 on October 20 and for version 0.3 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

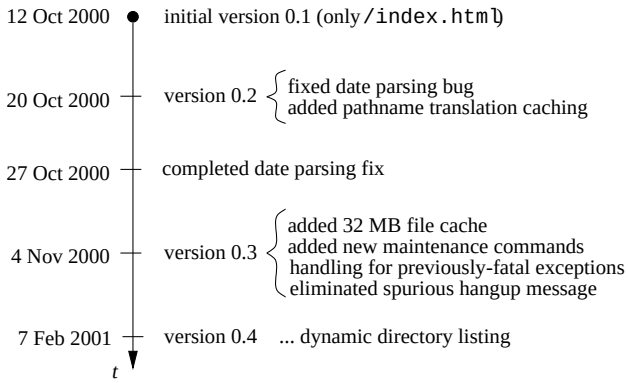

Figure 3: Timeline of *FlashEd* updates

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. As a result, we shut down the server on February 7[1] and redeployed it compiled with the new verifier. To accommodate these kinds of changes dynamically, 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 easy to use, since the only burden on the programmer is to fill out parts of the patch that the automated generator leaves out, and then to test the patches off-line. 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 quite valuable. 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 transformation 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.

[1]Actually, the power was accidentally shut off on the server, and so we took that opportunity to make the change.

6. PERFORMANCE ANALYSIS

Adding dynamic-updating imposes a number of costs on the system. At update-time, each patch must be verified and linked. At run-time, each external reference entails an extra indirection, essentially inherited from dynamic linking. 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 fully patched RedHat Linux 6.2, which uses Linux kernel version 2.2.17.

6.1 FlashEd runtime performance

The only runtime overhead of our implementation is incurred through the use of the GOT, which is inherited from dynamic linking. We could avoid this cost by implementing our linker to resolve external references *in-place*, as described in [11]. Using a GOT, each external reference requires two additional instructions, which adds about 2 cycles (or 6.7 *ns*) on our machines. By itself, this overhead is not very meaningful, since its overall effect is application-specific, depending on both the number of external function calls made during execution, and the amount of computation that occurs between those calls. To provide context, we examined the impact of this overhead on FlashEd's *application performance*.

To measure server performance, we used `httperf` (v0.8), which is a single, highly-parameterizable executable process that acts as an HTTP client. It can generate HTTP loads in a variety of ways, being able to simulate multiple clients by using non-blocking sockets. To ensure that the server is saturated, multiple `httperf` clients can be executed concurrently on different machines. Throughput is measured by sampling the server response over fixed intervals, and then summarizing the samples at the end of the test.

Our experimental setup is as follows. One machine runs the webserver, and the other three run `httperf` clients. To simulate 'typical' client activity, we ran a *log-based* test, in which each client uses an identical *filelist* containing a list of files to request, with a corresponding weight for each file. Each request is determined pseudo-randomly, as preferenced by its weight. Our filelist was obtained from the WebStone benchmarking system [23], which claims it to be a fair representation of file-based traffic. We used 90 second sample times, and we measured each server for roughly 32 minutes, totalling 21 samples. Because we observed skewed distributions in many cases, we report the median, rather than the mean, and use the quartiles to illustrate variability. (We had to make some minor changes to `httperf` to run this test.)

Figure 4 shows the results of our measurements. The X-axis varies with server version; the first three columns show the throughput for FlashEd 0.1, 0.2, and 0.3, respectively, and the fourth column shows the throughput for Flash (compiled using `gcc` version egcs-2.91.66 with flag `-O2`) as a point

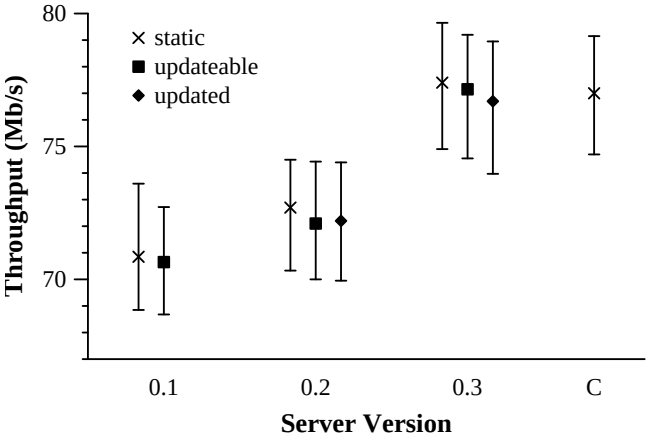

Figure 4: Flash and FlashEd throughput by version

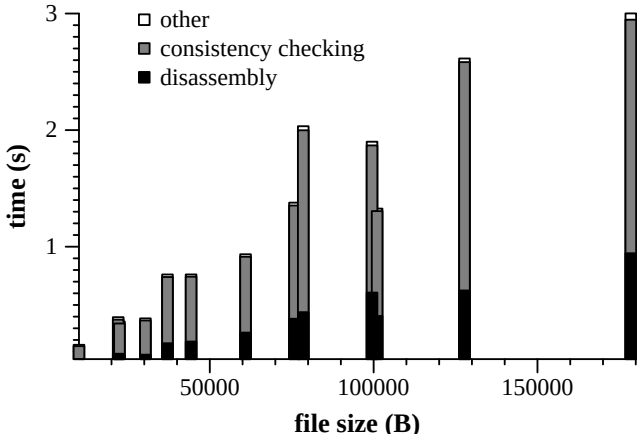

Figure 5: Time to apply dynamic patches

of reference. The Y-axis shows throughput in Mb/s (note that it does not start at 0). For each version of FlashEd, we measured the server's performance when it was compiled with and without updating support (labeled *static* and *updateable* in the figure, respectively), as well as when it was patched on-line (labeled *updated* in the figure); for example, the *updateable* FlashEd 0.3 was compiled directly from the version 0.3 sources, while *updated* FlashEd 0.3 was compiled from the version 0.1 and then patched twice dynamically. We suspected (correctly or incorrectly) that an updated FlashEd would have higher overhead than an updateable one due to a larger heap and memory footprint, since it retains the original version of the code in the text segment while new code is loaded into the heap. For each server we show the median throughput, with the quartiles as bars.

6.1.1 Analysis

The overhead due to updating is the difference in performance, per server version, between the medians of the static and updated/updateable versions. In all cases, this overhead is between 0.3% and 0.9%, which is negligible when compared to the measured variability. This variability is not unexpected because the URL request pattern seen by the server differs from sample to sample, since the URL's requested in aggregate by the three clients will differ during each sample (we chose longer sample times to mitigate this effect). To reduce experimental variability, we also ran tests in which the `httperf` clients constantly request the same URL, using a variety of URL file sizes. For these tests, the variability dropped significantly (the semi-interquartile range was typically 0.25% of the median, as opposed to about 3% for the log-based test), and the overheads were similar, ranging from 2.3% for a 500B file and 0% for a 500KB file; details on these tests can be found in [10].

The measurements do not consistently favor either the updated or updateable code. In particular, for FlashEd 0.2, the updated server is slightly faster than the updateable one, while the reverse is true for version 0.3. The fact that the relative and absolute locations of the code in an updated program is different than the updateable one may be one source of difference, since the same modules will be affected differently by cache policy. In addition, because the heap sizes are the same but the updated program uses some of this heap to store update code, we have observed that the updated code garbage collects more often, favoring the updateable code in this regard. However, in general this difference is well within the measured variability of the numbers and may be due to experimental variation.

We are encouraged by the fact that FlashEd 0.3 has performance essentially identical to that of Flash, though at first this is surprising, given our prototype compiler. However, much of the cost of file processing is due to I/O, reducing the benefit of compiler optimizations; a more CPU intensive task would certainly favor the C implementation. In any case, FlashEd's favorable performance suggests that TAL, and verifiable native code in general, is a viable platform for medium-performance, I/O-intensive applications.

6.2 Load-time overhead

Our updating system also imposes a load-time cost to link and verify dynamic patches. We measured the component costs to apply patch files, and in Figure 5 we present measurements for the patches to update FlashEd version 0.2 to version 0.3. All of these files are applied together, due to the mutually-recursive references among them, for a total time of 16.2s; 0.81s of this time was needed to relink the program.

In the figure, the X-axis is the total size of the compiled patch file, and each bar sums the total time to perform dynamic linking for that patch. We break down dynamic application into three operations: *disassembly* and *consistency checking*, which are the core of TAL verification, and *other*, which is the combination of the remaining costs, including the time to load and link the file, and to verify that its interface (the types of its imported and exported symbols and its exported type definitions) is consistent with that of the program. We can see that the total time is dominated by verification, averaging 72% for consistency checking and 25% for disassembly. According to [8], verification is generally linear in the size of the files being verified, which we find to be true here.

In many contexts, loading times of this magnitude are not a problem. For example, 16 seconds of pause time is less intrusive than an OS reboot. In the case of the webserver, an infrequent pause is at worse inconvenient to the user, but not harmful to the system. However, in other contexts, there may be good reason to want shorter update times.

We have identified three means of reducing the load-time

cost. First, the verifier could be further optimized, given that proof-carrying code [17] has demonstrated much smaller verification times, albeit with a different type system, and even TAL's implementors recognize that further gains could be made [8]. Second, verification could be performed in parallel with normal service, meaning that the system need only be stopped for linking and relinking, which have negligible cost. Finally, in the case of a completely trusted system (as is FlashEd, for example), we can safely turn off on-line consistency checking, running it instead for each loaded file on some other machine. Leaving on link-checking ensures that the loaded code meshes with the running program at the module level, but trusts that the contents of the loaded module are well-formed. Since consistency checking is the most time-consuming operation, we greatly reduce our total update times as a result.

7. 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, robustness, ease of use, and low overhead. A more complete discussion of related work may be found in [10].

7.1 Flexibility

At one extreme of the flexibility axis are systems that use dynamic linking alone to support updating [2, 21]. These solutions are only adequate when the programmer can correctly anticipates the form of future updates. Other systems are more flexible, but do not allow arbitrary changes. For example, Dynamic ML [7] only permits changing the definitions of types that are *abstract*, and updated modules cannot remove or change the types of existing elements. The Dynamic Virtual Machine [14], a Java VM with updating ability, and Dynamic C++ classes [12] similarly require class signature compatibility.

At the other extreme of the flexibility axis are systems that, like ours, allow nearly arbitrary changes to programs at runtime [13, 6, 9, 3, 4]. DYMOS [13] (DYnamic MOdification System) is the most flexible existing system; programmers can not only update functions, types, and data, but can also update infinite loops. Like ours, some past systems permit updates to active code. A gradual transition from old to new code occurs at well-defined points, such as at procedure calls [3, 13, 6, 4], or during object creation [12].

We believe our system sufficiently balances flexibility with the other updating criteria: the generality of our dynamic patches allows 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.

Pointerful Data. As mentioned in §3.1, we rely on the state transformer function to alter pointers to updated definitions that are stored in the program's data; such references could be to functions (*i.e.* function pointers), or to global data. For instance, when some function `f` is updated, a function pointer to `f` must be modified during state transformation to point to the new `f`; the system does not do this automatically. While handling 'pointerful data' in this way seems reasonable for imperative languages like C and Popcorn, this approach is likely insufficient for *functional languages* that make heavy use of closures (which are essentially function pointers), and may prove problematic for object-oriented languages as well.

An effective way to automatically update pointerful data is to selectively use reference indirection. In particular, we have experimented with having the compiler modify the code so that rather than passing or storing a function pointer, we pass the function's GOT entry. When the function pointer is actually used, the GOT entry is dereferenced, effectively retrieving the most recent version. This approach should apply equally well to pointers to data as well. We have largely implemented this idea, but still have a number of loose ends to tie up.

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 potentially prevents updating the abstract types, since it may be impossible to translate objects of the old type to ones of the new implementation.

We can solve this problem by implementing a more flexible type management policy. In particular, whether a type's implementation is visible or not to loaded code can be made a matter of policy; code that updates an abstract type may break the abstraction, but all other code may not. This should be reasonably easy to implement with our linker [11].

Unchecked updates. We currently support updates consisting of verifiable native code, but as motivated in §5.1.1, we would also prefer to occasionally update the *trusted* elements of the system, including the TAL verifier and runtime system. Implementing trusted updates should be straightforward by using the underlying loader without the verifier.

7.2 Robustness

Dynamic linking alone provides a significant advantage with respect to robustness 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 [2] 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.* [21] 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 our approach allows a programmer to determine when updates will occur, much work remains for determining *where* such safe points lie. In particular, things get more complicated with multithreading. Previous work [9, 13, 6] can serve as a starting point for this investigation.

Robustness 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 [14],

which makes use of Java bytecode verification. Other systems benefit from the use of type-safe source languages, like SML [2, 7], Haskell [21] and Modula [13], 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 [6, 9] and C++ [12], which lacks the benefit of type-safety.

7.3 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.

7.4 Low Overhead

Some systems provide updating at no runtime cost, including Gupta's system [9], and Dynamic ML [7]. Most systems employ reference indirection, either as we described in §3.2.1 [3, 14, 12], or in slightly more clever ways [13, 6]. Dynamic linking may (as in the case of ELF [22]) or may not impose an indirection, affecting systems like ours and those that use it exclusively [21, 2]. 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 [14].

8. 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, robustness, ease of use, and low overheads, as borne out by our experience with our dynamically updateable webserver, FlashEd.

Acknowledgments

We would like to thank the TALC group at Cornell University, and most especially Stephanie Weirich, Karl Crary (now at CMU), and Greg Morrisett, for the use and support of the TALx86 implementation, and for contributions to this work and work that led up to it.

9. REFERENCES

[1] The apache software foundation. `http://www.apache.org`.

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

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

[4] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, March 1983.

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

[6] 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.

[7] 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.

[8] D. Grossman and G. Morrisett. Scalable certification for Typed Assembly Language. In R. Harper, editor, *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2000.

[9] 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.

[10] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[11] M. Hicks, S. Weirich, and K. Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2000.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] M. Oehler and R. Glenn. HMAC-MD5 IP Authentication with Replay Prevention. Internet RFC 2085, February 1997.

[19] 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.

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

[21] 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.

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

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