Dynamic Software Updating

MICHAEL HICKS
University of Maryland, College Park
and
SCOTT NETTLES
The University of Texas at Austin

Many important applications must run continuously and without interruption, and yet also 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, ease of use, and low cost.

We present an 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 contain both 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 or Typed Assembly Language), which is native code accompanied by annotations that allow online 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, and generalize to present observations about updateable software development. Performance experiments show that for FlashEd, the overhead due to updating is low: typically less than 1 percent.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods; validation*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures; frameworks*; D.3.4 [**Programming Languages**]: Processors—*Compilers; run-time environments*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development; software maintenance*

General Terms: Design, Languages, Performance, Reliability, Verifications

Additional Key Words and Phrases: Dynamic software updating, typed assembly language

This work was supported by the National Science Foundation (NSF) under contracts ANI #98-13875 and ANI #0081360.
Authors' addresses: M. Hicks, Department of Computer Science, University of Maryland, A.V. Williams Building, College Park, MD 20742; email: mwh@cs.umd.edu; S. Nettles, The University of Texas at Austin, Electrical and Computer Engineering, 1 University Station C0803, Austin, TX 78712-0240; email: nettles@ece.utexas.edu.

1. INTRODUCTION

Many computer programs must be “nonstop”, 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 nonstop 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 online. This, of course, is not acceptable for nonstop applications; at best, it will result in loss of service and revenue, and, at worst, may compromise safety.

Thus, in general, nonstop 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 fifty-million-line transaction processing system; it is able to selectively take machines down and upgrade them by preserving relevant state in the online computers. This is similar to the way that Internet “server farms” are typically updated. This Visa system is updated many thousands of times per year, but tolerates less than 0.5% downtime [Pescovitz 2000]. 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.

While redundant hardware may often be present to support fault tolerance or load balancing, we 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, embedded control systems, and others.

Furthermore, there are many nonredundant systems that do not necessarily *require* nonstop 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. Finally, dynamic updating can aid the software development and maintenance process. For example, in our demonstration application, FlashEd, we dynamically inserted probes to observe the application’s state, and then removed them after it was reported. This was useful for debugging and profiling. Moreover, dynamic updating supports “fix-and-continue” methodologies of software development, so that fixes can be applied to the application while still in the state in which problems were observed.

We present a general-purpose framework for updating a program as it runs, 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.

We focus on the task of dynamically updating the code and state of a single process. Our work does not directly address the problem of dynamically updating distributed, cooperating programs. For example, we do not consider updating "web server farms" along with the underlying database they share. Changes to the database schema would require corresponding changes to programs that use the database, and these changes would have to be coordinated. Similarly, dynamically updating distributed programs to use a different protocol to use a newer protocol would require coordination to ensure that an updated program does not send a confusing message to a not-as-yet-updated one. Our framework can support changes to distributed programs that do not require coordination. For example, we can update a single web server operating within a farm. Because the state of the updated program is preserved, our approach may be preferable to "updating by load-balancing" in which the process is forced to shut down and restart after completing its transactions. In this case, it would lose performance-critical soft-state, like file and translation caches, while these caches are naturally preserved in our approach. Such systems would also benefit from the ability to add and delete debugging probes dynamically.

After stating the goals of dynamic software updating in Section 2, we describe our updating framework based on *dynamic patches* in Section 3 and our implementation of it using Typed Assembly Language [Morrisett et al. 1999b] in Section 4. We describe the development process for updateable software in Section 5, highlighting our tool for semi-automatically generating dynamic patches, and discuss the issues behind the *timing* of dynamic updates in Section 6. Section 7 presents our experience with a real-world application, a dynamically updateable web server called *FlashEd*; its performance is presented in Section 8. Section 9 discusses existing research and future directions.

A shorter description this work was published in the 2001 ACM Conference on Programming Language Design and Implementation [Hicks et al. 2001], and a complete description can be found in the first author's Ph.D. dissertation [Hicks 2001]. Compared to the conference paper, the present paper expands the exposition of the design, implementation, performance, and work related to our approach, while Sections 5 and 6, on the development process and timing of updates, are new.

2. GOALS AND APPROACH

What properties define an effective dynamic updating framework? To evaluate general-purpose dynamic updating systems, we establish four goals that any such system should ideally meet:

—**Flexibility.** *Any* part of a running system should be updateable without requiring downtime.

—**Robustness.** A system should minimize the risk of errors and crashes 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.

2.1 Existing Approaches

Unfortunately, no existing general-purpose updating system meets *all* of the desired criteria (an in-depth discussion of related work appears in Section 9). 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 [Appel 1994; Peterson et al. 1997] 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 [Gilmore et al. 1997] or named types [Duggan 2001], whole programs [Gupta et al. 1996], class instances [Hjálmtýsson and Gray 1998; Soules et al. 2003], or properly encapsulated objects [Boyapati et al. 2003]), *when* the updates can occur (e.g., only when updated code is inactive [Gilmore et al. 1997; Malabarba et al. 2000; Frieder and Segal 1991; Gupta et al. 1996; Soules et al. 2003]), or *how* the updates may occur (e.g., functions and values must not change their types [Hjálmtýsson and Gray 1998; Soules et al. 2003], or changes to module and class signatures are restricted [Malabarba et al. 2000; Gilmore et al. 1997]). 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 [Tool Interface Standards Committee 1995; Hjálmtýsson and Gray 1998; Frieder and Segal 1991; Gupta et al. 1996; Buck and Hollingsworth 2000] or have only dynamic checking [Armstrong et al. 1996], or require potentially error-prone hand-generation of complex patch files [Lee 1983; Gilmore et al. 1997; Malabarba et al. 2000; Frieder and Segal 1991; Gupta et al. 1996; Armstrong et al. 1996; Buck and Hollingsworth 2000]. Others rely on uncommon source languages or properties [Lee 1983; Bloom 1983; Armstrong et al. 1996; Boyapati et al. 2003] and hence are not broadly applicable. Finally, some systems impose a high overhead, either due to implementation complexities [Lee 1983; Buck and Hollingsworth 2000], or due to a reliance on interpreted code [Malabarba et al. 2000].

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) [Morrisett et al. 1999b]. 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. The construction of patches is largely automated and clearly separated from but compatible with 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 a substantial amount of useful patch code, leaving placeholders for the programmer in the (infrequent) nontrivial 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 run-time 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.

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.

It should be noted that while the framework itself does not restrict the timing of an update, there will undoubtedly be moments when updates should not occur, dictated by the semantics of the program. For example, updating the representation of a map from a tree to a hashtable should likely not occur while the map is in use. On the other hand, updating the code that manipulates the map to perform extra logging or to cross-reference another datastructure to improve lookup times is likely acceptable even while the map is in use. Our system allows the programmer to make such determinations, while nonetheless ensuring the program remains type-safe. Some restrictions on update form (Section 3.4) may effectively induce restrictions on timing as well. The issue of timing is discussed at length in Section 6.

```
static int num = 0;
typedef struct {
  int a; int b;
} t;
int f (t T) {
  num++;
  return T.a + T.b;
}
```

Fig. 1. Example file `main.pop`.

3. DYNAMIC PATCHES

Our approach to dynamic updating is built on the idea of a *dynamic patch*, which describes the dynamic changes between two versions of a program module. How we define a dynamic patch influences both the system's flexibility and its ease of use: it should ideally be able to express arbitrary changes to a file, and it should cleanly separate constructs required for patching from the new code, allowing the software development process to be cleanly separated from patch development. It also affects the system's robustness, as implementing the patch semantics could be quite difficult, resulting in a large and/or complex implementation. We present our notion of dynamic patch incrementally, arriving at a definition that is suitably flexible, all the while keeping the new code separate from code germane to patching.

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 version of the file and S is an optional *state transformer function*, used to convert the existing state accumulated by f to a form usable by the new code f'. The transformer is defined such that the old and new code have their own state, and thus the old state is copied to the new code and then properly transformed. This notion of patch is similar to Gupta [1994], except that he defines essentially a single patch for the entire program, instead of one patch per changed file.

In our system, patches may be used to reflect nearly arbitrary changes to a file dynamically, particularly to its function and data definitions, and its type definitions. We look at each of these cases in turn, and then describe some of the limitations of our patch definition.

3.1 Changes to Code and Data

As an example, consider the module shown in Figure 1.[1] The function `f` increments `num` to track the number of times it is called and returns the sum of the two fields of its argument, which has type `t`. Suppose we modify `f` to return the product of its arguments, creating a new version `main.pop`*(2)*. The dynamic

[1]The code examples shown here are written in *Popcorn*, a safe C-like language [Morrisett et al. 1999a]. We've made some small modifications to Popcorn syntax in the figures, so the reader can think of this code as essentially C. We discuss Popcorn further in the next section.

new version main.pop*(2)*:

```
static int num = 0;
typedef struct {
  int a; int b;
} t;
int f (t T) {
  num++;
  return T.a * T.b;
}
```

state transformer S

```
void S () {
  main.pop(2)::num =
    main.pop::num;
}
```

Fig. 2. Dynamic patch for main.pop: (main.pop(*2*), S).

patch that converts main.pop to main.pop*(2)* is shown in Figure 2.[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'.

Of course, the transformer function is arbitrary code, so more complicated transformations can be expressed when needed. For example, say the variable x has type graph, which represents a directed graph. The module containing x is updated, but the representation of type graph is unchanged, so state transformation can simply copy the reference to the new module: x = Old::x. Alternatively, say the representation of type graph changes to include an additional field at each node. In this case, the programmer could perform a deep copy of the graph by traversing the old graph and generating new nodes containing the old information plus the new field. Auxiliary datastructures (like hash tables) can be used to keep track of the aliasing relationships in the old graph to ensure they are duplicated in the new one.

On the other hand, if the program depends on the actual addresses of pointer data, an acceptable state transformer will be more difficult or impossible. For example, say the addresses of graph nodes are used as keys in a separate hash table. If the representation of graph changes and we must copy it as above, we must likewise transform the hashtable to use the copied keys. We further consider the limitations of state transformation in Section 3.4.1.

3.2 Stub Functions

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 referrers to changed items will access them at the old (now incorrect) type. For example, consider a new version of f that adds a new argument to f (call the new file version main.pop*(3)*); existing callers in different modules will still call f with a single argument, constituting a type error. 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.

On the other hand, in some situations a changed file's callers cannot be changed. For example, in some proposed *active networks* [Tennenhouse et al.

[2]This figure illustrates an abstract notion of a patch; the actual syntax for our implementation is presented in Section 4.3.

new version `main.pop`*(3)*:

```
static int num = 0;
typedef struct {
  int a; int b;
} t;
int f (t T, int x) {
  num++;
  return T.a * T.b + x;
}
```

state transformer S

```
void S () {
  main.pop(3)::num =
    main.pop::num;
}
int stub_f(t T) {
  return f(T,0);
}
```

Fig. 3. Dynamic patch for `main.pop`: (main.pop(*3*), S, {`f` → `stub_f`}).

1997], multiple parties may download code into routers to customize packet processing. In this case, one party may wish to update their code, but cannot update the callers of that code belonging to other parties. As another example, we may wish to break down a large update into several smaller updates, so that the process of updating the system is less disruptive; Lee [1983] considers a way of methodically breaking down larger updates into smaller ones. For these situations, we allow patches to include *stub functions*. A stub function has the same type as the old version, and is interposed between old callers and new definitions to get the types right.

The patch for `main.pop` to `main.pop`*(3)* is (main.pop(*3*), S, {`f` → `stub_f`}), shown in Figure 3. The third part of the patch is a mapping between functions in `main.pop` and the stub functions that should replace them. In this case, the stub function for `f`, called `stub_f`, simply inserts a default value for the new argument `x` to `f`. Existing callers of `f` will now call `stub_f`, while code loaded later will link against the new `f`, at the new type.

Even when all patches are applied at once and/or the types of updated functions do not change, stub functions can be used to perform incremental, transitional computation. For example, suppose we have an event-based system. Every time the program is prepared to process an event, it calls the function `get_next_event`, which returns a value of type `event`. Now, say we wish to update the way that events are gathered; that is, we wish to change the implementation of `get_next_event` to perform some additional, or different operations. Assume that `get_next_event` keeps a queue of events waiting to be processed.

We could write a patch for this change in one of two ways. We could write a state transformer function S to copy the contents of the old queue of events for use by the new `get_next_event`. Alternatively, we could use a stub function to retrieve the queued events using the old function, and then switch to using the new one when no old events remain, roughly:

```
static bool got_old_events = false;
event stub_get_next_event() {
  if (!got_old_events)
    event e = Old::get_next_event(); /* old version */
    if (e != null) return e;
    else got_old_events = true;
  }
  return get_next_event();           /* new version */
}
```

Here, we differentiate between the old version of get_next_event and the new one by prepending the old version with Old::. Using a stub in this way deals with the old state incrementally, as opposed to performing all transitional computation at patch-time. The obvious benefit is that the pause at patch-time due to state transformation is less; this may be critical to reduce service outage when transforming large amounts of state.

On the other hand, existing code will always call the stub function, even after all of the old state has been processed, imposing extra overhead. To avoid this cost, we could define special syntax to allow a stub function to update its clients to point the actual function, rather than the stub, once the state transformation is complete. The above code would change to something like:

```
event stub_get_next_event() {
  event e = Old::get_next_event(); /* old version */
  if (e != null) return e;
  else {
    RELINK("get_next_event",get_next_event);
    return get_next_event();       /* new version */
  }
}
```

The call to RELINK would cause existing clients to call the new version of the function, rather than the stub, on subsequent calls. How this construct would be implemented would depend on the mechanisms used to realize dynamic updating, which we discuss in Section 4.3.

There is no obvious construct for data analogous to stubs. Thus, if a patch changes the type of some global variable, then all the functions in the running program that refer to that variable must also be changed. In the case that global data is declared static, no additional files are involved since only the functions in the local file itself are affected. When data is exported, however, all other files that refer to that data must be updated. Note that this problem is mitigated in object-oriented languages, since data is co-located with the code that operates on it.

3.3 Changes to Type Definitions

Changes may also occur to type definitions which declare the named types of the program. Considering again the code in Figure 1, we could change the definition of t to include a third field. A simple way to express a changed type in a patch file is to syntactically differentiate between the type's old definition and its new one, so that we can manipulate data inhabiting both types. A patch in this style is shown in Figure 4. Here the stub function stub_f takes an argument having the old type t, syntactically shown as having type Old::t. It then creates a value of type t, copying the existing fields from the argument, and assigning a 0 for the third field. Finally, the new f function is called with the newly created value.

Differentiating between the old and new version of types allows the program to have values of both the old and new t during its execution. The benefit here new version `main.pop`(4_1):

```
static int num = 0;
typedef struct {
  int a; int b; int c;
} t;
int f (t T) {
  num++;
  return T.a * T.b * T.c;
}
```

state transformer S

```
void S () {
  main.pop(4)::num =
    main.pop::num;
}
int stub_f(Old::t T) {
  t T2 = new t(T.a,T.b,0);
  return f(T2);
}
```

Fig. 4. Dynamic patch for `main.pop`: (main.pop(4_1), S, {`f` $\rightarrow$ `stub_f`}).

is that dealing with data of changed type is completely *programmer-directed*, which allows more relaxed constraints on when updates can be performed at run-time, and is more predictable. Programmer-directed patches admit a reasonably simple implementation.

3.4 Limitations

Our notion of patch is designed to be flexible, but it has some limitations to ensure a simple implementation. In particular, there are fundamental limitations on the process of state transformation, the patch definition does not provide a means to deal with data on the stack, and its requirement that programmers manually handle existing data may sometimes prove burdensome; we consider each point in turn.

3.4.1 *Expressiveness of State Transformation.* There is a fundamental assumption that the form of the new state can be determined from the existing state, implemented by the programmer as the state transformation function S. For example, converting a tree to a hash table is straightforward because the *information* contained in the tree is sufficient to construct the hash table. However, new data structures may contain information not available in the current program. For example, we could imagine adding a time stamp field to a data structure T to indicate when it was created. Dynamically transforming existing T instances to add this new field will be impossible because the information simply does not exist at dynamic-update time. We call this the *state transformation problem*. Bloom and Day [1993] have explored some of the theoretical limitations of state transformation.

This problem can be overcome by modifying the code and data structures in question to handle the lack of information. In the worst case, the programmer could use a version field to differentiate between data-deficient structures converted at update-time and those created by the new code. In the places that it matters, the deficient data structures can be treated specially, perhaps by calling bits of old code, as we did for the queue example in Section 3.2. In our experience, the state transformation problem rarely arises; we describe the sole case we have encountered in Section 7.1.1.

Others have handled the state transformation problem by allowing old code and new code to be intermixed [Appel 1994; Segal and Frieder 1993; Hjálmtýsson and Gray 1998; Duggan 2001; Bierman et al. 2003], and the choice of which to execute to be determined automatically. Existing data instances are *not* transformed, and are used exclusively by older versions of the code. Newly created instances are always of the most recent version, and are handled by the new code. Unfortunately, this semantics does not lead to correct program operation in general, and thus the programmer is forced to ensure an orderly transition. However, it becomes quite difficult to do so given the possibility of many versions of code and state interacting together. We feel our approach keeps things simpler, while still providing enough control. More experience is needed to understand how often this process is burdensome in reasonable cases, and/or whether the "pollution" of the source code to accommodate one-time patches will be excessive.

3.4.2 *Transforming the Stack.* The state transformation function S considers global state, including the heap and static data segment, but not the stack. As a result, there is no direct means for the programmer to transform data on (or pointed at from) the stack. In contrast, systems like Dynamic ML [Gilmore et al. 1997] and others [Boyapati et al. 2003; Duggan 2001] can automatically transform all data having the changed type from wherever it may be reached.

Preventing direct manipulation of the stack has two consequences. First, old code and data may, for a time, be active along with new code. This is advantageous in that there is no need to translate the return address of the running code to return instead to the new version of its caller. If the caller changed significantly, it would be difficult to decide where to return instead. On the other hand, having two versions of a function or data active in the program may lead to incorrect and/or unpredictable behavior. One way to handle multiple versions is to use stub functions. That is, when old code calls new functions, it will do so through the stub functions. These functions could attempt to ensure a consistent semantics, but admittedly determining reasonable behavior could quickly become quite complicated in even simple situations.

The second consequence is that the implementation burden is reduced. In particular, there is no need to support a general way of traversing the stack. Type-safe, runtime stack traversals would require that extra semantic information be available at runtime, such as that needed by a debugger. Adding such a mechanism in a system like TAL would also increase its trusted computing base.

We opted for the simpler implementation at the cost of some reduced flexibility. As we describe in Section 6, to avoid dealing with the stack at patch time, we constructed our application to only permit updates when the stack was essentially empty. This required slightly restructuring the application. Recent work is starting to explore type-safe ways of supporting garbage collectors, thread schedulers, security checkers, etc. It would be interesting to revisit this issue at that point and experiment with possible approaches.

3.4.3 *Transforming Existing Data.* While our style of programmer-directed patch is simple and intuitive, it nonetheless places an added burden on the programmer. An alternate approach is *system-directed* patches, in which only one version of a type's data can be present at any given time, as enforced new version `main.pop`(4_2):

```
static int num = 0;
struct t {
  int a; int b; int c;
}
int f (t T) {
  num++;
  return T.a * T.b * T.c;
}
```

state transformer S

```
void S () {
  main.pop(4)::num =
    main.pop::num;
}
Old::t convert_t(t T) {
  t T2 = new t(T.a,T.b,0);
  return T2;
}
```

Fig. 5. Alternative notion of dynamic patch for `main.pop`: (main.pop(4_2), S, {}, {`t` → `convert_t`}).

by system. In this case, programmers define *type conversion functions* in the patch to indicate how data of the old type should be transformed to the new, and the system invokes these functions when needed. This approach has the advantage that state transformation can occur *on demand*, reducing potentially-long pauses when the program is updated. Using this semantics, our patch file might be like that shown in Figure 5.

In this case, there is no need to explicitly define a stub function as we did in Figure 4. The system will properly convert the data using `convert_t` as needed. As such, the programmer is only responsible for *how* the data is converted, not *when*. System-directed patches are employed by Dynamic ML [Gilmore et al. 1997] and others [Duggan 2001; Boyapati et al. 2003].

The disadvantages of system-directed patches are twofold. There is a greater implementation burden: since data must be tagged with its type, there must be some means for finding and converting the data. In addition, erroneous conditions (such as exceptions thrown during transformation) may be difficult to handle. Furthermore, a system-directed, automatic approach to converting old data may be too inflexible for certain applications. In particular, to ensure soundness, there are restrictions on when type conversions can take place. For example, Boyapati et al. [2003] develop a system for lazily upgrading objects in a persistent store. For their system to work properly, the programmer must properly use language-level *transactions* to avoid conflicts, and be sure that data encapsulation properties imply a well-defined ordering. Without transactions, the changes must be transparent to the surrounding code, since it will be unaware of when the change is to take place. Transparency typically implies that changed types must be *abstract* [Gilmore et al. 1997].

We favor programmer-directed data conversion because of its greater flexibility and simpler implementation, with a smaller trusted computing base. Our experience with FlashEd has been that manual conversion is not difficult. However, the notion of system-directed transformation is appealing both from a usability and performance point-of-view. We leave it to important future work to discover ways of combining the best of both approaches.

In summary, our preferred definition of dynamic patch is $(f, S, \mathit{stub\ set})$, where f is the new code, S is the state transformer function (acting on the heap and static data), and *stub set* is a set of mappings from functions to their corresponding stubs. This notion is both flexible and easy to use, as it can reflect nearly arbitrary changes between two files, and the code germane to dynamic patching is cleanly separated (as the state transformer and stubs) from the new code. The fact that we can generate patches mostly automatically, as described in Section 5, further bolsters our claims of ease of use.

4. IMPLEMENTING DYNAMIC PATCHES

There are two basic ways we could implement dynamic patches. The first would be to compile the new program from the new source files, start it up alongside the old one, and then signal the old one to marshal and transmit its state to the new program. The new program would unmarshal and transform this state, and initialize its execution using the result. We call this approach *state transfer*-based updating. The alternative is to dynamically link patches into the existing program, transform the state locally, and then transition to using the new code. We call this *dynamic linking*-based updating.

The state transfer approach has been implemented by Gupta and Jalote [1993]. It is appealing in that we can upgrade a program and its underlying hardware at once by starting the new program on a different machine. However, it has a number of drawbacks:

—Old and new code cannot run concurrently, even temporarily, which limits when updates might occur.
—An update will always affect the entire program. For example, all of its state must be transferred from the old process to the new, even if only a fraction of it is actually changed by the update.
—State transfer is more challenging to implement, since machine state like the program counter, heap, stack, etc. must be reified correctly in the new program, which may have a wholly different layout. Indeed, to address this point Gupta only permitted the updating of global variables that were scalars; transforming linked structures was not supported. Work platform-independent checkpointing could probably be applied to address some of these concerns [Ramkumar and Strumpen 1997].
—Some program state may be stored in the operating system kernel, like file descriptor maps or parent-child process relationships, and this may not be easily captured and moved between the old and new process.

Therefore, for reasons of flexibility and simplicity, we build dynamic patch application on top of dynamic linking. In this section, we first consider possible mechanisms for transitioning the program to use dynamically linked patches, considering updates to code and data, and then updates to type definitions. We conclude with the details of our implementation.

4.1 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. This is shown in Figure 6. By contrast,

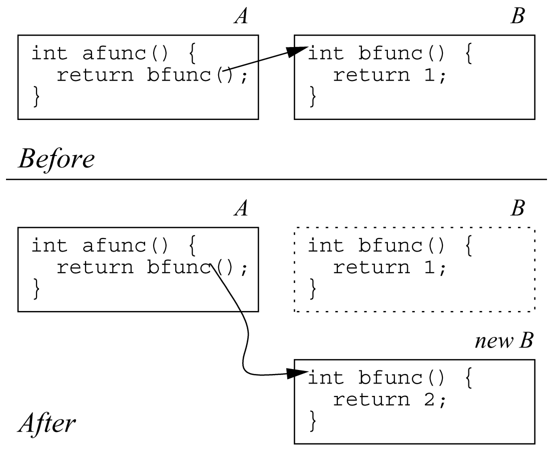

Fig. 6. Updating by code relinking.

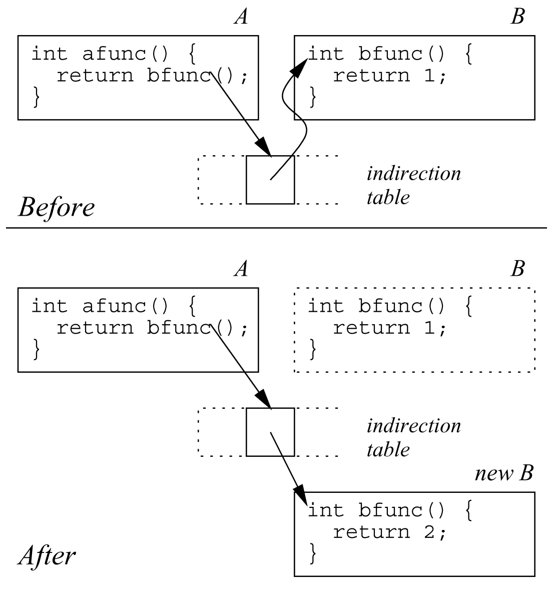

Fig. 7. Updating by reference indirection.

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. This is shown in Figure 7.

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

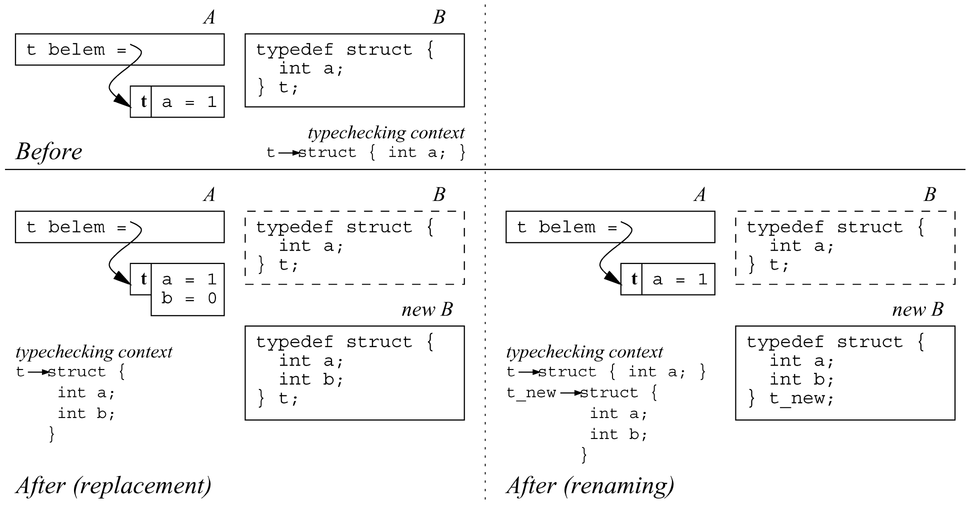

Fig. 8. Two methods of updating type definitions: *replacement* and *renaming*.

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 Section 4.3.1). One 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. Relinking may also incur a higher update-time cost, since all callers to a changed function must be rewritten, as opposed to a single table entry. In our experience this cost has been negligible.

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 Section 9.

4.2 Updating Type Definitions

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. This idea is illustrated in the left side of Figure 8. Here, a new version of module *B* has been loaded that updates the type definition for `t`. The notion of `t` is updated in the typechecking context, and the existing instance of `t` is converted to a new one (the new field is filled in with a default value). Because the implementation of `t` has changed, any code that makes use of `t` elsewhere in the program must itself be replaced. One exception is in the case that 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.

The type renaming approach is shown in the right side of Figure 8. Now, the new version of *B* defines a different type `t_new` as the logical replacement of `t`. Existing instances of `t` 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 old definition of `t` and adds the new one for `t_new`.

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 [Gilmore et al. 1997], or else requires a way of converting from the new version back to an old one [Duggan 2001], which can make the program harder to reason about. 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 run-time 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 [Oehler and Glenn 1997]) 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 [Frieder and Segal 1991; Lee 1983; Gupta et al. 1996]. Recent systems mitigate these problems by discovering changed objects lazily [Boyapati et al. 2003; Duggan 2001], and/or by using indirection [Soules et al. 2003] to effectively update all aliases.

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 [Malabarba et al. 2000] 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 Section 7.1.1.

4.3 Prototype Implementation

We have implemented our framework to target Typed Assembly Language (TAL) [Morrisett et al. 1999b]. Both TAL and its cousin, proof-carrying code [Necula 1997], 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 [Morrisett et al. 1999a], includes a TAL verifier and a prototype compiler from a safe-C language, called Popcorn, to TAL.

4.3.1 *Dynamic Updating.* We provide dynamic updating for Popcorn programs by extending the functionality of TALx86's type-safe dynamic linker [Hicks et al. 2000]. We briefly describe the original 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 [Tool Interface Standards Committee 1995]. 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 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 transfers the old state from the old to the new *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 GOTs 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 [Hicks 2001].

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 not strictly correct because this code might still 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 nonweak pointer elsewhere in the program, and thus code can be collected when it is no longer needed.

Unfortunately, TAL does not 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.

4.3.2 *Patches.* Our implementation of dynamic patches closely follows the abstract description of Section 3. 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: the new 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. An example patch file is shown later, in Figure 12.

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.

4.3.3 *Compiler Optimizations.* Global state is transformed at the time of an update. Therefore, we require that compiler optimizations not cause writes to globals to be reordered before or after an update point, and that any globals allocated in registers be flushed back to main memory. This turns out to be easy, since a program update point is simply a call to a well-defined function the compiler can be made aware of. That is, in our system, a program updates itself at a well-defined moment, rather than being interrupted and updated at an unexpected time. We describe our timing model further in Section 6.

The use of indirections in the GOT to facilitate relinking implies that any updateable function should not be in-lined. If some calls were in-lined, those calls would not notice dynamic updates. We have not found this to be a performance problem with I/O-intensive applications, as detailed in Section 8. A possible solution is to employ technology as in Sun's HotSpot JVM [Java 2002], which un-inlines code when debugging. In our case, we would un-inline functions that needed to be updated.

5. GENERATING PATCHES

Given a means to dynamically patch running programs, we must now consider how best to generate these patches. A novel aspect of our approach is the mostly automatic generation of patch files. This feature was originally designed to make the system easier to use during the development of FlashEd, our updateable webserver: it is very tedious to write state transformation and stub functions by hand. It has also proven invaluable in minimizing human error, since it is less likely that a necessary state transformation or stub function will be accidentally left out: it discovers all files that have changed, and leaves *placeholders* in the generated code where programmers need to fill in the details. As it turns out, a very simple syntactic comparison of files, informed by type information, can do a good job of identifying changes and partially generating patch code.[3] In this section, we discuss how patch generation fits into the normal software development process, explain the patch generation algorithm itself, and then present some examples of its use.

5.1 Software Development for Dynamically Updateable Systems

A typical way to develop software is as follows: Each version of a program is given a revision number, and the corresponding program source is associated with that revision, probably archived with revision control software. When changes need to be made, such as to fix bugs or add new features, the current version is modified to effect those changes. Once these changes have been thoroughly tested, the modified source is assigned a new revision number, archived, and deployed. In short, to make a software change, we start with the current source, modify and test it, and then deploy the changed program as the new version.

Our approach to building dynamically updateable systems alters this process only slightly. Just as before, programmers make changes to the current sources, and then compile and test the result to create the new version. Once the new version is stable, rather than halting the existing version and then deploying the new one, patches are created that reflect the differences between the old and new versions of the software. In our system, much of these patch files can

[3]Note that our patch generator is similar in spirit to the *transformGen* database schema evolution system developed by Garlan et al. [1986].

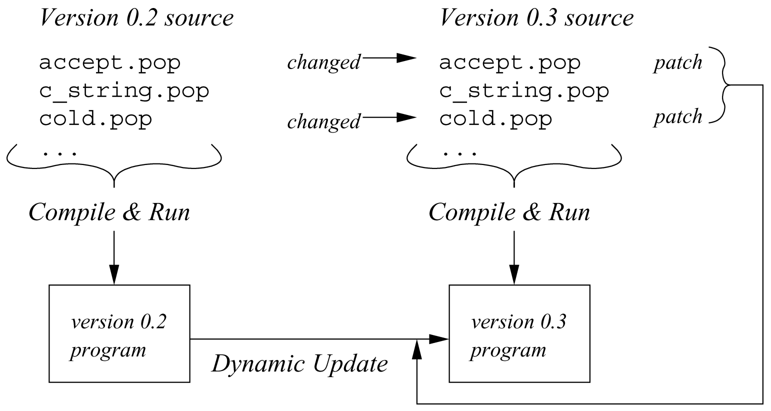

Fig. 9. Building and maintaining an updateable program.

be generated automatically. The programmer only fills in the parts of the state transformer and stub functions that cannot be automatically generated. The patches are then dynamically applied to the old version of the software, thereby migrating it to the new version.

The development process is depicted in Figure 9. The current version 0.2 of some software consists of a number of source files. In moving to the next version, 0.3, many of these files have changed. When testing is complete, patches are created for the changed files. These patches are then dynamically applied to the currently running version 0.2, resulting in a running program equivalent to version 0.3. The new version retains the state of the old version, and only negligibly interrupts (but does not cancel) service while the patches are applied. These are the benefits of dynamic updating; if we were to instead shut down the old version and restart with the new one, the running program's state would be lost, and any midstream processing would be forcibly cancelled.

Our methodology cleanly separates software development from patch development. Such a separation is possible because our notion of patch (and our implementation of it) is cleanly separated from the software itself. Moreover, because our notion of patch is quite flexible, the process of development is not hampered by what may be expressible as a patch. In many other systems, patches are limited to certain forms, and so software development is similarly limited. For example, in Dynamic C++ classes [Hjálmtýsson and Gray 1998], only changes to *instance* methods and data may be reflected dynamically; per-*class* (i.e., `static`) methods and data cannot evolve. As a result, new static methods must be *added* to programmatically replace the old ones, but the old ones will remain, cluttering the code and obscuring its meaning.

On the other hand, there are times when writing a valid state transformer is not possible without further altering the source files. For example, it may be that the existing state respects one invariant, but the new version of that state respects another. If the old state cannot be transformed to respect the new invariant, the new code could be changed to temporarily accept state having the old invariant, until it is no longer needed. In our experience, such changes are rare; we consider this issue more when describing our experience with FlashEd.

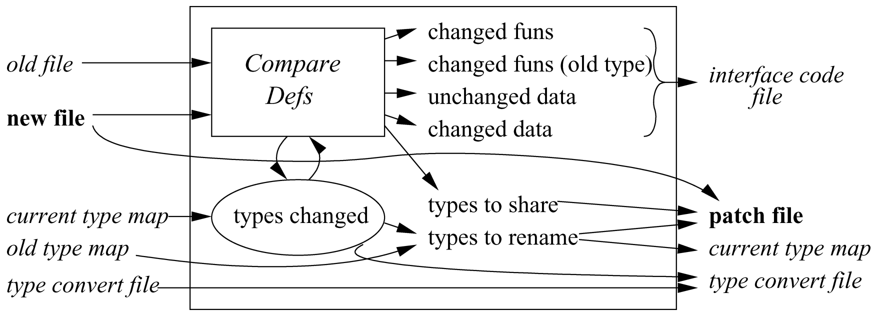

Fig. 10. Structure of the automatic patch generator tool.

5.2 Automatic Patch Generation

The schematic of our patch generator is illustrated in Figure 10. As inputs, the patch generator takes the new file, the old file, a current *typename map*, the old file's *typename map*, and the current *type conversion* file. Only the new file is required, all other arguments are optional. The results of patch generation are the patch file, the interface code file, the updated typename map, and the updated type conversion file; if the new and old files differ then a patch file will always be generated, but the other outputs are generated only if needed.

Patch generation is broken into two stages, *identification* and *generation*. The identification phase is shown in the figure as the *Compare Defs* box. It takes the old and new files as inputs, along with a set of named types that are known to have changed. The set starts off as empty, or may be initialized by the contents of the current *typename map* file. The algorithm works as follows. First, the old and new files 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; if found to be different, the name of the type is added to the set of changed types. 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. Furthermore, a function is considered to have changed if it references static data or functions that have themselves changed; the reason for this is explained shortly.

The results of the identification phase are a number of sets that describe the differences between the files. These sets are (1) the functions that changed, (2) the functions that changed but retained their old type, (3) the data declarations that *did not* change, (4) the data declarations that changed, (5) the named types that *did not* change, and finally (6) the named types that did change. These sets are used, along with some of the inputs, to generate the patch file and some supporting files, including the interface code file, the type conversion file, and the typename map file. We cover each of these in turn.

Interface code file. The interface code file contains the state transformation function `S`, and any needed stub functions. To construct `S`, the toplevel contents of unchanged global variables are simply copied, either as a variable assignment, as shown for `num` in Figure 2, or with element-wise copying for arrays and aggregates. When a value's type changes, the old value must be converted to the new type during the copy. For named types whose definitions have changed, we generate *type conversion functions*, and call those; these are described below. We perform assignments for like primitive types (e.g., promoting a `int` to a `float`), but otherwise insert a "placeholder" indicating where code should be inserted by hand.

The patch generator also generates default stubs for functions that have changed type. Two basic modes are possible. In the simplest mode, the generator creates a function body having the old type, but which raises an exception. This mode is useful when all patches for the running program are to be applied simultaneously, in which 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, as shown in Figure 4.

Typename map file. 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 (which includes the type name), meaning that the same definition will always generate the same name. This allows development of patches by multiple programmers without the worry of choosing incompatible type names.

The mapping from old to new name is stored in the *typename map* file. This file is read in as each patch is generated, so that the global fact that a type changed informs the local process of patch generation for a particular file. The updated map file is written out upon generation completion. Crucially, the typename map file for the old version will also be consulted so that types that have changed name as a result of earlier patches are properly named in the current set of patches.

Type conversion file. Finally, *type conversion functions* are constructed for data conversion from old to new versions of a named type, and vice-versa. These are used by the interface code files, as mentioned above. All type conversion functions are stored in a separate file; this file is read in at the start and written out upon patch generation completion, with new conversions functions for changed types not already covered. Eventually, the type conversion file is dynamically loaded into the running program along with the other patches for their use.

For `struct` types, unchanged fields are copied, and new fields are initialized with a default value. Each field that has changed type is translated, either by calling another type conversion function, or by a local translation, inserting 'placeholders' where translations are unknown. Popcorn's tagged `union` types are translated by deconstructing the value of the old type, and reconstructing a new type by cases, following the above rules.

The patch generator assumes a correspondence between the names of types in the old version and in the new. Therefore, if the new version changes its name, old version `old_foo.pop`:

```
typedef struct {
  int a; int b;
} *t;
t someTs[];
int f (t T) {
  return T.a + T.b;
}
```

new version `new_foo.pop`:

```
typedef struct {
  int a; int b; int c;
} *t;
t someTs[];
int f (t T) {
  return T.a + T.b;
}
```

Fig. 11. The old and new versions of example file `foo.pop`.

then the patch generator will think of this as a new type. This difficulty of changing names arises in other areas, in particular file synchronization [Tridgell and Mackerras 1996; Balasubramaniam and Pierce 1998]; we should be able to apply its solutions to our patch generation system. For example, some problems could be alleviated by permitting the user to inform the generator of *old* $\rightarrow$ *new* relationships between definitions having different names. Instrumentation of the development environment could also be used to generate such relationships.

5.3 Example

To illustrate the patch generator in action, consider the following example. Figure 11 illustrates the old and new versions of some file `foo.pop`. The new version has changed in two ways: the type of the structure `t` has changed to include an additional field `c`, and as a result the function `f` has now changed type, since it takes a value of the new type `t`, rather than the old `t`. Providing these two files as input to the patch generator results in four output files, shown in Figure 12. They are the patch description file `new_foo.patch`, the interface code file `new_foo_patch.pop`, the typename map file `TYPENAME_MAP`, and a file containing the type conversion functions, `convert_patch.pop`.

The patch generator observes that the type `t` changed, so it generates a name for the new version of `t` from the MD5 hash of its definition. It stores this mapping in the `TYPENAME_MAP` and indicates it in the `renaming` list of the patch description file. The `TYPENAME_MAP` file should be used as input for other patches in the same program that reference `t`, so that even if those files do not change themselves, they will be considered to have changed since the definition of `t` is different.

The interface code file `new_foo_patch.pop` defines a state transformer function `S` to translate the array `someTs`, and a stub function for `f`, since `f` now takes a value of the new type `t`. For the array transformation, a loop is generated that piecewise translates the elements of the array by calling the type conversion function `t__old2new`; this function is defined in `convert_patch.pop`, explained below. Note that the loop refers to the `someTs` array using the prefix `New::`. As explained in Section 4.3.2, this prefix ensures that the new version of the array is stored into, since the lack of a prefix defaults to the old version.

The stub function simply raises an exception indicating that an existing caller has not been properly updated. Note that the array conversion is not entirely correct: the new version of the array `someTs` needs to be allocated before the copying can take place. Retaining more information during the new_foo.patch:

```
implementation: new_foo.pop
interface: new_foo_patch.pop
renaming:
  New::t=MD5(
    typedef struct {
      int a; int b; int c;
    } *t)
```

TYPENAME_MAP:

```
New::t=MD5(
  typedef struct {
    int a; int b; int c;
  } *t)
```

new_foo_patch.pop:

```
#include "core.h"
typedef struct {
  int a; int b;
} *t;
extern t someTs [];
extern New::t t__old2new (t);
static void S () {
  int idx__0;
  for (idx__0 = 0;
       idx__0 < size(someTs);
       ++idx__0)
    New::someTs[idx__0] =
      t__old2new(someTs[idx__0]);
}
prefix Stub {
  int f (t v0) {
    raise (new Core::InvalidArg(
      "Stub?f (int (New::t))"));
  }
}
```

convert_patch.pop:

```
typedef struct {
  int a; int b;
} *t;
typedef struct {
  int a; int b; int c;
} *New::t;
New::t t__old2new (t from) {
  if (from == NULL)
    return NULL;
  else {
    New::t to = new New::t{b=from.b,
                           a=from.a,
                           c=0};
    return (to);
  }
}
t t__new2old (New::t from) {
  if (from == NULL)
    return NULL;
  else {
    t to = new t{b=from.b,a=from.a};
    return (to);
  }
}
```

Fig. 12. The patch and supporting files generated for `foo.pop`.

identification phase concerning how globals are allocated, either statically or dynamically, would allow this transformation to be more precise.

Finally, the file `convert_patch.pop` contains the type conversion functions for translating values to and from the old and new versions of `t`. As with interface code files, the old and new versions are differentiated by their prefix: new versions are prepended by `New::`, and old versions have no prefix. Note that in this case, a default value of 0 is generated for the added field; we could conceivably have inserted a comment to remind the user to use a more appropriate value if necessary.

The function `t__old2new` translates an element from the old type `t` to the new one. When the function is called, a new value of type `t` is allocated and initialized with the fields it shares with the old value. Since the new `t` has an added field, the patch generator also inserts a default value for that field. Note that this function does not preserve aliasing relationships, since a new instance is allocated each time. If aliasing is important, this problem is easily solved using a hashtable to map old pointers to their new versions. In the example, this could be done in the `S` function in `new_foo_patch.pop`. The loop that converts the `someTs` array would maintain the hash table, and only call `t__old2new` as necessary. Since it is not clear that preserving aliasing is always necessary, and adds cost to the transformation, we leave it to the programmer to insert such code when needed.

The function `t__new2old` translates in the reverse direction, dropping the value in the new field. In general, functions x`__old2new` are useful in state transformation, while x`__new2old` functions are useful in stub functions, for returning an value of old type to an existing caller.

6. WHEN TO APPLY PATCHES

So far we have concentrated entirely on *how* dynamic updates can be realized, and *what* well-formed updates will consist of. However, an equally important question is *when* updates should be performed. To understand the question of timing, and why it is important, we consider two models of updating, the *interrupt model* and the *invoke model*. The essential difference between the two is that in the invoke model, the conditions under which an update may be performed are determined *statically* (at compile-time), while for the interrupt model they are determined *dynamically* (while the program is running). While many dynamic updating approaches use the interrupt model for its apparent increase in flexibility, we argue that it makes determining appropriate update times no less difficult, and requires greater implementation complexity than the invoke model, which is our model of choice.

6.1 Interrupt Model

In general, it is possible for a well-formed update to be applied at a bad time, resulting in incorrect state. For example, consider our very first example, 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* (i.e., `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`.

In part, the above scenario occurs because we assume that a program could be updated at any moment during its execution. This implies an *interrupt*-driven model of updating: the program is interrupted at some point during its execution, the update takes place, and then the program is resumed. This model can be more controlled. Rather than performing the update at the moment of interruption, the update can be delayed until certain conditions are satisfied. These conditions might be determined automatically, if possible. For example, some systems forbid updates to modules that are *active* [Gilmore et al. 1997; Malabarba et al. 2000; Segal and Frieder 1993; Soules et al. 2003; Boyapati et al. 2003]. On-line upgrading support for the K42 operating system [Soules

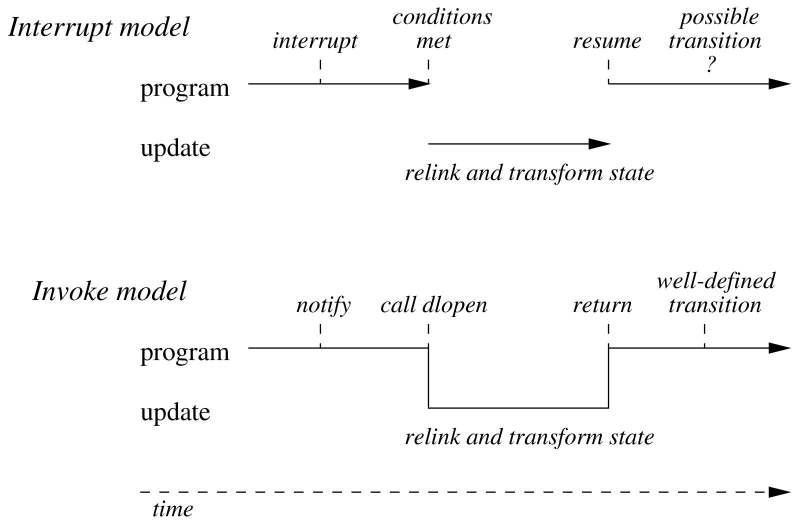

Fig. 13. Two models for updating a single-threaded program.

et al. 2003] supports delaying an object upgrade until that object is inactive. In both cases, the transition to new code always occurs immediately upon program resumption.

Alternatively, conditions can be provided by the programmer. For example, in DYMOS [Lee 1983], the programmer specifies *when-conditions* along with the patches to update as in

```
update P, Q when P, M, S idle
```

This specifies that procedures `P` and `Q` should be updated only when procedures `P`, `M`, and `S` do not have activations in any thread stack.

This *interrupt model* is visualized in the top portion of Figure 13. During its execution, the program is interrupted, then after some time the necessary conditions are satisfied and the program context-switches to perform the update atomically.[4] Control then returns to the running program, and at some point the program *transitions* to using the new code. For example, if procedure `Q` was running when the update took place, the old `Q` would continue to run and the new `Q` would be invoked sometime later.

Being able to enforce timing conditions at runtime adds flexibility, since only the program state at the time of the update need be considered. However, it lacks *predictability*, since it is not clear when or if such timing conditions will ever be satisfied. Even specifying those conditions is not necessarily straightforward. In fact, Gupta et al. have shown that the problem of correct timing is, in general, undecidable. To show this, they developed a formal model for dynamic updating and defined a notion of *update validity* [Gupta 1994; Gupta et al. 1996].

[4]This does not mean that the update cannot be performed in parallel with program execution, although this is frequently the case in existing systems, only that the update must *appear* atomic to the program.

Because no automated means of generally determining a valid update time is possible, previous researchers have developed techniques to identify program patterns that have valid update points. Gupta et al. developed an algorithm that compares the old and new versions of C code (not including functions, stack allocation, or heap allocation) and identifies, based on a syntactic analysis, program points that would preserve update validity. This analysis is quite conservative, and can only handle restructurings of the same algorithm, not changes to program functionality. Lee [1983] describes a way to decompose a valid update into a set of smaller valid updates. A directed graph is constructed such that each node in the graph represents a function to be replaced, and an edge from f to g implies that *g should be updated before or with f*. The strongly connected components of the graph then represent functions that must be updated together. Lee does not formalize why one procedure should be updated before another; in some cases this is easy to determine (e.g., if the types of functions change), but in others it is not straightforward. Furthermore, a valid update must be known before it can be deconstructed, but no guidance is provided in finding such an update. Many systems simply impose the restriction that updates may only occur to inactive code [Gilmore et al. 1997; Malabarba et al. 2000; Segal and Frieder 1993; Soules et al. 2003], but this does not guarantee that race conditions will not occur.

Enforcing arbitrary when-conditions at runtime can be expensive, in terms of performance and implementation complexity. For the system to test update constraints at runtime, there must be some way to identify the set of active procedures. If some procedure required to be idle is in the set, then the program continues to execute, updating the active set as it goes, with each change to the set testing whether the idle conditions have been met. PODUS [Frieder and Segal 1991] relies on the fact that its programs must be single-threaded and therefore the active set is effectively the stack; the set is maintained by extra code that checks the stack depth upon procedure return. In DYMOS, which supports multi-threading, each function call requires a synchronized access to some global structures to store the fact that the function is active; this can be quite costly.

Whether the problems we have described with identifying valid dynamic timing constraints arise in practice is uncertain. However, few of the systems mentioned above present any analysis or experience that says otherwise. Therefore, we are led to believe that while flexibility is *potentially* increased by timing enforcement mechanisms, using these mechanisms may or may not result in actual gains, calling into question the loss in performance and predictability and the increase in implementation complexity.

6.2 Invoke Model

The problem of timing can be greatly simplified by specifying update points *statically* rather than dynamically. That is, rather than assuming, as in the interrupt model, 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 by *invoking* a special `update` procedure. This model, which we call the *invoke model*, is illustrated in the bottom half of Figure 13. Here, the program is somehow notified that it should perform an update. The next time it invokes the `update` procedure, the waiting patches are applied, and the program continues where it left off. If it was properly constructed, it should transition to the new code at a well-understood time.

6.3 Example

Based on our experience, we have found a general approach to structuring applications so that updates are well-timed when using our system. This is not the only possible program structuring, but we believe it works well.

The problem with arbitrary update times is two-fold:

(1) Running procedures might be manipulating state we want to transform; the interaction between the state transformer and these procedures could result in race conditions. To prevent this, we essentially want to delay state transformation until running code has completed *transactions* manipulating global state. The notion of transaction has been formalized in the database community as being a series of operations that must occur all-at-once (as far as the rest of the program is concerned) or not at all. Since most programming languages (including Popcorn) do not support formal transactions, we consider a more informal notion. In particular, a program transaction is a computational sequence that performs some self-contained piece of work.

(2) We want the transition to the new code to be well defined. If functions have activation records on the stack, and these functions are updated, then the old code will run until the functions exit and are re-entered. Unless the program is structured in a reasonable way, running functions may not exit (and re-enter the new code) in a timely fashion, leading to potential problems. For example, old code could continue to operate on old copies of global state, thus not properly communicating computation to the new code.

We can solve both of these problems by requiring that the program *unwind* the stack at update-time. That is, any code that is currently executing must exit, without performing any meaningful (i.e., state-manipulating) computation. Once all active functions have exited, the program restores the stack to its former state by calling into the new code, and resumes its computation on the transformed state. This way, any piece of code that was executing, even an infinite event loop, can be updated in a timely manner.

The program's transactions must be identified when implementing stack unwinding. In particular, updates should only be applied when there are no active transactions. This allows the stack to be safely unwound and restarted, since all meaningful work has been completed. Event-based programs are easily restructured to unwind and restart computation. In particular, each event-handler essentially implements a transaction, so that an update notification can be processed at the start of the event loop, when there are no transactions being processed. The loop can then be exited and restarted, using the new code.

We take this basic approach with FlashEd, as we will describe in detail in the next section.

It should be noted that "internal" transactions relevant to updating need not correspond to external transactions the entire program might be involved in. For example, an event-driven web server will likely process a single HTTP request using many internal transactions. One transaction parses the request and queues an event to retrieve the requested file; the next several transactions read that file in chunks; and the final few transactions transfer the file to requestor. The state of the overall HTTP request is kept consistent after each internal transaction, so an update could be performed then. Thus, updates can occur in our framework while the program is actively processing requests (and indeed this is the case with FlashEd).

This approach can also apply to multithreaded programs by employing barrier synchronization. First, each thread can be notified that an event is pending. The threads then complete any outstanding transactions, and unwind their stacks. They 'check in' with the main program thread, at which time the update is applied. Finally, the main thread notifies the remaining threads that they may restart, at which point they begin using the new code. While simple and effective, this approach exacerbates the possibility that an application will be unresponsive while it is being updated, since many threads may be waiting for others to synchronize. The programmer must therefore ensure that `update` points will be reached often enough to mitigate this possibility. We further note that even if the program does become unresponsive for a time, this is much preferable to the interruption that would occur if it actually had to be stopped and restarted.

6.4 Discussion

The fact that the invoke model fixes the moment(s) of update is both an advantage and a disadvantage. On the one hand, our ability to reason about an update's correct timing is increased because we know *exactly* when updates will occur and thus can determine how they will interact with the system. On the other hand, choosing update times may be difficult since they must accommodate updates of unknown composition, and if an update time is chosen poorly, we may be limited in the updates we can perform correctly (at least until we can update the program to accept updates at other times). However, our experience with the program structure described above, which derives from our approach in FlashEd, has been that *a priori* choosing a reasonable update time has not affected what changes a patch can express, and has greatly improved our confidence in its correctness. That is, the advantage of fixed timing is significant and the disadvantage of fewer times for update is minimal.

To be fair, determining proper update timing to ensure both responsiveness and correctness is the largest unexplored area of this work. We have had good results with single-threaded, event-driven programs, but have little practical knowledge in the way of multi-threaded programs. We are encouraged that updating multi-threaded programs is not onerous with the invoke model, as this model is successfully employed by multi-threaded Erlang [Armstrong et al.

1996] programs. We believe there is ripe opportunity to apply formal methods to proving that an update is valid. One possibility is to use formal transactions to support proper update timing [Bloom 1983; Boyapati et al. 2003]. We discuss this approach in more detail in Section 9.

7. 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 [Pai et al. 1999]. Flash consists of roughly 12,000 lines of C code and has performance competitive with popular servers, like Apache [2001]. We constructed our version, called *FlashEd* (for *Ed*itable *Flash*), by porting Flash to 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.

7.1 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 nonstop 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.

Aside from changes due to porting from C to Popcorn, these were the only two changes we had to make to FlashEd to support dynamic updating.

7.1.1 *Patching.* To gain experience evolving a program using our system, we constructed FlashEd *incrementally*. Our initial implementation (version 0.1)

Table I. Summary of Changes to Versions 0.2 Through 0.4 of FlashEd

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

lacked some of 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 Section 5.1. 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 I. 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, so functions in otherwise unchanged 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 occasionally had to fill in placeholders in the generated type conversion functions, particularly for function pointers (like a 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.

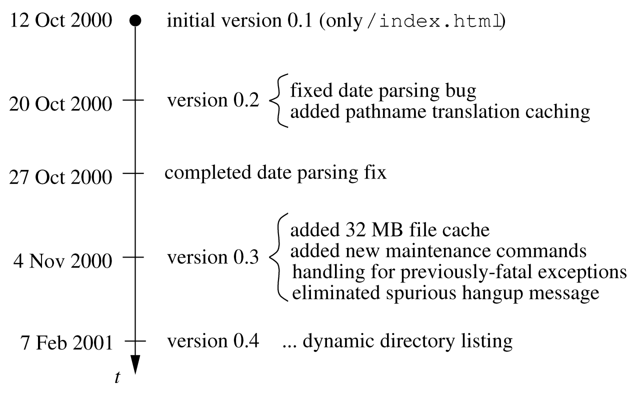
12 Oct 2000
initial version 0.1 (only /index.html)
20 Oct 2000
version 0.2
fixed date parsing bug
added pathname translation caching
27 Oct 2000
completed date parsing fix
4 Nov 2000
version 0.3
added 32 MB file cache
added new maintenance commands
handling for previously-fatal exceptions
eliminated spurious hangup message
7 Feb 2001
version 0.4
... dynamic directory listing
t

Fig. 14. Timeline of *FlashEd* updates.

As mentioned in Section 3.4.1, our approach may require changes to the source program to solve instances of the state transformation problem, in which existing state cannot be straightforwardly converted to allow newer code to process it. For FlashEd, this situation arose in the patch from version 0.2 to version 0.3, which adds a file cache. In all versions of FlashEd, the datastructure used to maintain connection state contains a field `dataEnt` which has struct-type `DataEntry`. Elements of type `DataEntry` contain information about the file that a particular URL refers to, including its modification date, size, contents, and so on. In versions earlier than 0.3, this entry was constructed from scratch for each request and then thrown out after the connection closed. In version 0.3, this entry is inserted into the file cache after it is first created, and then shared among concurrent connections that have requested the same file thereafter. The difficulty is that when the connection is complete, the code for releasing `DataEntry` values expects these values to respect certain invariants. Constructing the entries out of context to respect these invariants is not trivial.

Therefore, we had to slightly change the source of version 0.3, specifically the routine `ReleaseDataEntry`, to deal specially with those `DataEntry` values that are problematic. This would prevent them from confusing the invariants maintained by the file cache software. Fortunately, the changes to be made were fairly simple in this case, but it may be that more complicated changes would be necessary in other cases. While our experience has largely been that software and patch development are separate processes, this particular case suggests that in general the development process for updateable software is an iterative one: develop the next version of the software and test it; develop the patches and test them; if during patch development any changes needed to be made to the source, go back and test the static program.

7.2 Experience

To simulate a production environment, we ran a public FlashEd server with the intention of never shutting it down, making all changes online. A brief chronology for FlashEd is shown in Figure 14. 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 online 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 revealed a number of interesting practical aspects. For instance, we learned soon after we deployed the server that our version of the TAL verifier was buggy—it only checked 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 and redeployed it compiled with the new verifier.[5] 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 was 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 offline. 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 in order 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.

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

8. PERFORMANCE ANALYSIS

Supporting 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.

8.1 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* [Hicks et al. 2000]. 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) [Mosberger and Jin 1998], 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.

8.1.1 *Log-Based Test.* To get a sense of overall server performance, we ran a single server on one machine, and one `httperf` client on each of three other machines creating "typical" traffic patterns. Each client's requests were determined by an identical *filelist*, which consists of a list of files to request, with a corresponding weight for each file. Each request is determined pseudo-randomly, based on its weight. Our filelist was obtained from the WebStone benchmarking system [Webstone 2001] as a fair representation of file-based traffic. It is shown in Figure 15. The first column indicates the URL and the second column indicates the weight to assign to it; the comment in the third column indicates the file's size in bytes.

We used 90 second sample times, and we measured each server for roughly 32 minutes, totaling 21 samples. Because we observed skewed distributions in

`/file500.html`	350	#500
`/file5k.html`	500	#5125
`/file50k.html`	140	#51250
`/file500k.html`	9	#512500
`/file5m.html`	1	#5248000

Fig. 15. Filelist used in the log-based test.

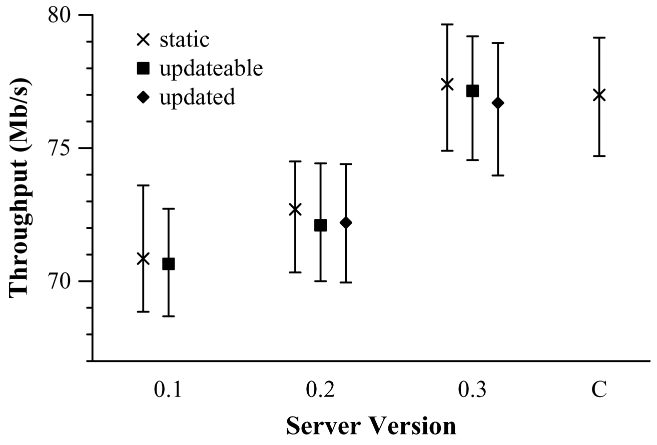

Fig. 16. Flash and FlashEd throughput by version.

many cases, we report the median, rather than the mean, and use the quartiles to illustrate variability. With 21 samples, the interval between the bars serves as a 98% confidence interval [Pratt and Gibbons 1981]. Note that we had to make some minor changes to `httperf` to run this test.

Figure 16 shows the measured throughput. 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 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 online (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.

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).

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.

8.1.2 *URL Test.* The log-based test characterizes "typical" performance, but has two shortcomings. First, the actual activity seen by the server is variable from sample to sample, since the URLs requested in aggregate by the three clients may differ during each sample. We extended the sample time to 90 seconds to mitigate this problem, but each value in Figure 16 has a (relatively speaking) sizable variability. In particular, the *semi-interquartile range* (SIQR), which is the difference between the high and low quartiles divided by two, is roughly 3% of the median, as compared to a SIQR of $<1\%$ of the median for the tests we are about to describe. The second problem is that the log-based test provides less of a sense of "worst-case" overhead, because some of the files requested during the test are quite large, and thus the I/O time dominates the overhead imposed by updating. To address the problem of per-sample variability, we ran tests that request the same URL repeatedly. To examine how I/O dominates updating overhead, we considered a variety of URL file sizes, from 500 B files to 500 KB files. For each URL, we used a 10 second sample time, and ran each test for just over 5 minutes, totaling 31 samples. Again, we calculated the median and the quartiles.

The results are shown in Figure 17, which has the same format as Figure 16. The first thing to notice here is that the variability is much decreased; in particular the SIQR is typically less than 0.5% of the median, and except in the case of 500k files, the interquartile ranges rarely overlap for each cluster of points. The range of the Y-axis differs for each graph, with none starting at 0. The error bars for the 500k files have the same size as the rest but appear more significant because the scale of the Y-axis is smaller.

To understand the trends exhibited in this graph, we graphed the overhead due to updating, shown in Figure 18, where the X-axis is URL file size—shown for 500 B, 1 KB, 10 KB, and 500 KB files—using a logarithmic scale for presentation purposes, and the Y-axis is percent overhead from the nonupdateable (static) FlashEd. There are three basic trends:

(1) The overhead for updateability decreases as the size of the file increases. For the 500-byte file, we see as much as a 2.3% overhead, while for the 500-kilobyte file the overhead is 0. This is most likely because the added I/O time overwhelms the extra processing cost.

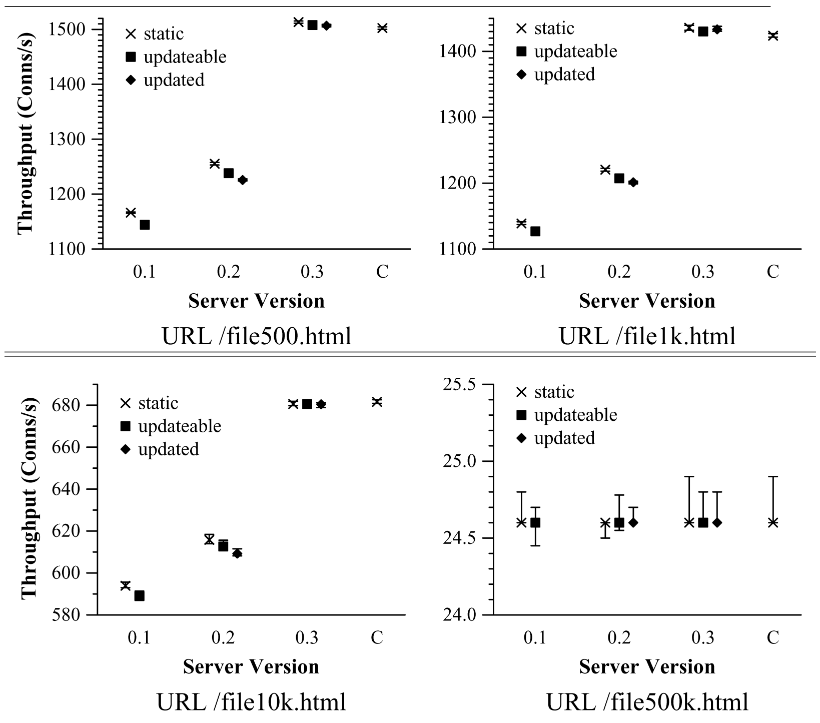

Fig. 17. Flash throughput for URL-based tests.

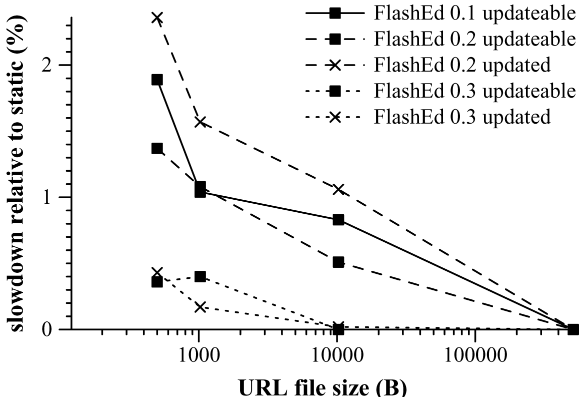

Fig. 18. Correlating the overhead of updateability with URL file size.

(2) In general, the relative overhead due to updating decreases as the version number of FlashEd increases. This can be seen in the figure by comparing all of the "updateable" lines (whose points are marked with boxes) and comparing all of the "updated" lines (whose points are marked with diamonds).

There are two explanations for this phenomenon. First, because the processing time per request decreases for each file, while the network transfer

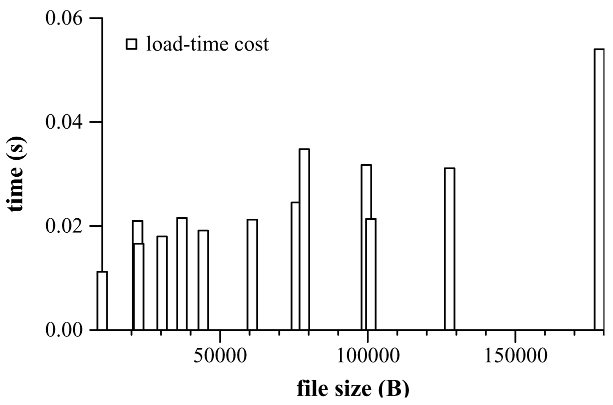

Fig. 19. Time to apply dynamic patches (link-checking only).

time remains the same, the impact of updating is decreased. Second, there are fewer external references made for version 0.3 than for versions 0.1 and 0.2. Because the runtime penalty of an extra indirection occurs only when references are to definitions not in the current file, the fewer these kinds of references, the lower the overhead. 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 rebuilt the three versions of FlashEd and ran our tests on each of them. For versions 0.1, 0.2, and 0.3 of the server, the percentage of dynamic references to external definitions was 70%, 71%, and 62%, respectively, meaning that version 0.3 incurs a lower penalty for indirection, relative to its nonupdateable version.

(3) The relative overhead of updated and updateable versions is inconsistent; that is, sometimes the updated version performs better and sometimes the updateable one does. This follows the same pattern as the log-based test.

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.

8.2 Load-Time Overhead

Our updating system also imposes a load-time cost to apply the dynamic patches. Figure 19 shows the time to apply patches that update FlashEd version 0.2 to version 0.3. Each bar in the figure represents a single file, where the X-axis indicates the total size of the compiled patch file, and the Y-axis is the time to apply the patch. Each bar includes the time to (1) dynamically load the file, (2) perform *link-checking*, and (3) link the file into the current program. Link-checking is the process of verifying that a loaded file's interface (the types

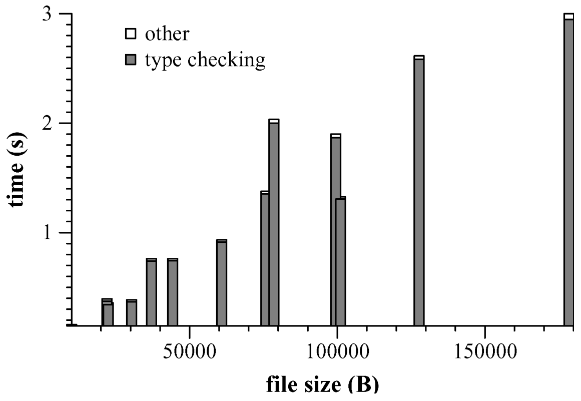

Fig. 20. Time to apply dynamic patches (link-checking and type-checking).

of its imported and exported symbols and its exported type definitions) is consistent with that of the running program. All of these files are applied together, due to the mutually-recursive references among them, for a total time of 0.33 seconds. The cost to relink the program and run the state transformers after all files are loaded in this case was an additional 0.81 seconds, for a total of 1.14 seconds.

Only a negligible portion of the relinking time includes state transformation. For this update, most of the state in updated files was unchanged, and so was simply transferred by reference. The contents of two arrays changed type, so they needed to be copied. The first was the connection handler array, indexed by file descriptor, and the second was the connection array, indexed by connection number, containing the state of outstanding connections. Both the handler and connection types changed, and so existing entries needed to be copied and transformed. In this experiment, these arrays were empty since no requests were being serviced, so we only paid the cost of allocation and initialization of the arrays in the new code. In total, roughly 100 global variables were transferred by reference, and a few hundred KB were allocated for the new arrays.

The times in Figure 19 do not include the cost of type checking. We assume the owner of the non-stop application is the only one allowed to update it, and therefore typechecking can be performed off-line. In contrast, performing link-checking online ensures that the loaded code meshes with the running program at the linking level, which is important for ensuring safety. Indeed, linkchecking caught the error reported in Section 7.2. Were we to perform typechecking online, it would not add to assurances of safety, and would result in a significant pause, as shown in Figure 20. The figure adds the cost of typechecking to the costs illustrated in Figure 19 (labeled as "other"). This includes the cost to disassemble the object file, and the cost to type check the disassembled representation. On average, typechecking costs 75% of the total time, with the remaining time for disassembly. According to Grossman and Morrisett [2000], verification is generally linear in the size of the files being verified, which we find to be true here. If for some reason type checking should be performed online, it could be performed in parallel with normal service, meaning that the system need only be stopped for linking and relinking, which have negligible cost.

9. 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 the first author's Ph.D. dissertation [Hicks 2001].

9.1 Flexibility

At one extreme of the flexibility axis are systems that use dynamic linking alone to support updating [Appel 1994; Peterson et al. 1997]. These solutions are only adequate when the programmer can correctly anticipates the form of future updates and structures the program to accommodate them. This is because dynamic linking only allows new code to be plugged into existing interfaces. Other systems are more flexible, but do not allow arbitrary changes. For example, Dynamic ML [Gilmore et al. 1997] 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 [Malabarba et al. 2000], a Java VM with updating ability, Dynamic C++ classes [Hjálmtýsson and Gray 1998], and K42 [Soules et al. 2003] similarly require class signature compatibility.

A number of systems support "fix-and-continue" development, in which program code can be updated while it runs. This feature was typical in Common Lisp and Smalltalk implementations. More recently, Sun's HotSwap JVM permits dynamic, type-compatible changes to method bodies [Java 2002; Dmitriev 2001]. The DynInst system [Buck and Hollingsworth 2000] similarly allows the dynamic addition and removal of machine code. Such systems generally focus on updating the *code* of running program, but have little or no support for updating its *state*. This works for small changes, or for the insertion of instrumentation or debugging code, but hampers long-term dynamic evolution.

At the other extreme of the flexibility axis are systems that, like ours, allow nearly arbitrary changes to programs at runtime [Lee 1983; Frieder and Segal 1991; Gupta et al. 1996; Armstrong et al. 1996; Bloom 1983]. DYMOS [Lee 1983] (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 [Armstrong et al. 1996; Lee 1983; Frieder and Segal 1991; Bloom 1983], or during object creation [Hjálmtýsson and Gray 1998].

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, which we will describe next.

Pointerful Data. As mentioned in Section 3, 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 may be 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 *object-oriented* languages, whose objects can viewed as records of closures.

Object-oriented systems like Dynamic C++ classes and K42 use added indirection to make updating object references automatic. The former uses a proxy class to wrap an underlying object; the proxy and the object share the same interface. All calls to the proxy's methods are forwarded to the underlying object. However, when a new version of a class is loaded, only newly created objects use that version, while existing objects continue to use the old version. K42 essentially implements the reference indirection approach we describe in Section 4, by employing a global indirection table for all object references. When an object is updated, its entry in the global table is changed, so that all aliases are simultaneously updated. The added indirection adds to performance overhead for each dereference, but has been shown to improve multiprocessor performance scalability [Gamsa et al. 1999].

Another option would be a hybrid approach using reference indirection for pointerful data, and code relinking for direct calls. 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 and to data. We implemented this idea for function pointers, but found that it can interact poorly with polymorphism [Hicks 2001], and so continue to use manual pointer updating by default.

Distributed Programs. As mentioned in the introduction, our approach does not directly address the problem of coordinating distributed processes that should be updated together, for example, because a shared communication protocol must be altered, or because the representation of shared state, such as that stored in a database, must change. Work on distributed updates [Magee et al. 1989; Kramer and Magee 1990; Hofmeister 1993] considers the problem of coordination. Sewell [2001] has explored means to ensure distributed programs behave properly when communicating values of possibly different versions of an abstract type. Our work could profitably be combined with these to allow both updates within a process as well as a means to coordinate those updates across processes.

We believe an interesting avenue of future work is to combine the problem of database evolution with the problem of code updating. In object-oriented databases, where code and data are combined as object instances, this combination is straightforward and is being actively explored [Boyapati et al. 2003]. However, the problem of updating a relational database along with the programs that are actively using it has remained relatively unexplored. Some work has considered database *views*, which allow programs to view the same database as having different schemas, but this approach hampers longer-term evolution since changes must always be backward-compatible. We believe that we can build on our approach of synchronizing on `update` points in multi-threaded programs to address this problem, using static analysis to ensure safety.

9.2 Low Overhead

Some systems provide updating at no runtime cost, including Gupta's system [Gupta et al. 1996], and Dynamic ML [Gilmore et al. 1997]. Most systems employ reference indirection, either as we described in Section 4.1 [Armstrong et al. 1996; Malabarba et al. 2000; Hjálmtýsson and Gray 1998; Soules et al. 2003], or in slightly more clever ways [Lee 1983; Frieder and Segal 1991]. Dynamic linking may (as in the case of ELF [Tool Interface Standards Committee 1995]) or may not impose an indirection, affecting systems like ours and those that use it exclusively [Peterson et al. 1997; Appel 1994]. As we have demonstrated, 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 such as in the DVM Malabarba et al. [2000].

Most approaches, including our own, require that relevant data is transformed at the time of an update. However, if the application is maintaining a very large dataset, this transformation could create a long, undesirable pause in processing. An alternative is to delay state transformations until relevant data is actually used. Two recent approaches support such lazy updates. Duggan [2001] permits objects of *named type* to be updated lazily. Each time data of named type is accessed, it must be *opened*, at which point the compiler provides the *version* of the data it expects. If the runtime version is different, then a version transformer (or a composition of them) runs to convert the data to the expected type. This is similar to, but more flexible than, our use of stubs (Section 3.2), using the model of type replacement. The problem is that data must support both upgrading and downgrading, which may not be possible. Moreover, the programmer has a more difficult time reasoning about the overall correctness of a program, since the point of transformations, and whether they go forward or back, is not under manual control.

Boyapati et al. [2003] support a lazy update system for persistent objects. Like Duggan, objects are not updated until they are used. However, downgrades can be avoided by forcing an update ordering that avoids having new objects received by existing code. A type system that enforces encapsulation is able to ensure that all of an object *o*'s encapsulated objects are updated first, so that *o*'s state transformation function can assume its fields have already been updated. The programmer has some control over update timing through the use of program-level, database-style *transactions*. For example, if an update to an object is noticed while the object is in use, the current transaction is aborted, and the object is updated before the transaction restarts. The drawback of this approach is that strict encapsulation boundaries must be drawn so that the approach can be fully automatic.

We also note that Dynamic ML [Gilmore et al. 1997], which performs its updates using a modified garbage collector, could perform updates incrementally or concurrently by using a more advanced collector. A key disadvantage of lazy updates is that complicates handling errors that could arise during state transformation. In particular, in our approach, if a state transformer throws an exception, the entire process can be immediately rolled back. In a lazy approach, some of the state may already have been transformed and manipulated by the program before the error is discovered, leaving the program in an unrecoverable, corrupt state.

9.3 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 [1994] 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. [1997] 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 Section 6, 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. As mentioned above, Boyapati et al. [2003] use encapsulation and first-class transactions to ensure safely-ordered updates, even with multithreading. However, determining appropriate transaction points is quite similar to our requirement of finding safe update points. K42 uses a protocol for ensuring that an object is *quiescent* before it is updated [Soules et al. 2003]. The idea is to block threads from entering an object that needs to be updated, and wait until all current threads have stopped using the object. Assuming that the object properly encapsulates its state (there are no aliases to it directly), and that all individual method calls are essentially complete transactions, the object can be updated at that time, and the blocked threads restarted in the new object. In a sense, this is a particular implementation strategy to ensure a transactional semantics. The problem is the lack of automatic support for determining when the given assumptions are reasonable, and the failure to support more general transactions (say, across multiple method calls). We believe a more general formal understanding of application-specific update correctness is needed. We and others have done some formal work [Bierman et al. 2003; Gupta et al. 1996; Lee 1983; Frieder and Segal 1991] toward this end.

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 [Malabarba et al. 2000], which makes use of Java bytecode verification. Other systems benefit from the use of type-safe source languages, like SML [Appel 1994; Gilmore et al. 1997], Haskell [Peterson et al. 1997] and Modula [Lee 1983], 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 [Frieder and Segal 1991; Gupta et al. 1996] and C++ [Hjálmtýsson and Gray 1998; Soules et al. 2003], which lacks the benefit of strong typing.

9.4 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. We are currently working on newer tools to do a better job of automated state transformation. In particular, we are exploring how understanding a program's refactorings (semantics-preserving changes), as opposed to functional changes, can help in automatically generating a state transformation function, particularly with relation to how like data is moved between types, or variables in different files.

10. 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 anonymous reviewers for their constructive and helpful feedback. Thanks to the former members of TALC group at

Cornell University, most especially Stephanie Weirich, Karl Crary, 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. Thanks also to Jonathan Moore, for his early contributions to this work, and to Benjamin Pierce, Jonathan Smith, Insup Lee, and Mark Segal, who served on the first author's dissertation committee and provided valuable feedback and suggestions.

REFERENCES

Apache. 2001. The Apache Software Foundation. http://www.apache.org.

Appel, A. 1994. Hot-sliding in ML. Unpublished manuscript. http://www.cs.princeton.edu/~appel/papers/hotslide.ps.

Armstrong, J., Virding, R., Wikstrom, C., and Williams, M. 1996. *Concurrent Programming in Erlang*, Second ed. Prentice-Hall, Englewood Cliffs, N.J.

Balasubramaniam, S. and Pierce, B. C. 1998. What is a file synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*. Dallas, TX. ACM, New York (Oct.) 98–108. (Full version available as Indiana University CSCI technical report #507, April 1998.)

Bierman, G., Hicks, M., Sewell, P., and Stoyle, G. 2003. Formalizing dynamic software updating. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*. Warsaw, Poland (April).

Bloom, T. 1983. Dynamic Module Replacement in a Distributed Programming System. Ph.D. dissertation, Laboratory for Computer Science, The Massachussets Institute of Technology.

Bloom, T. and Day, M. 1993. Reconfiguration and module replacement in Argus: Theory and practice. *Softw. Eng. J.* 8, 2 (Mar.), 102–108.

Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H., and Richman, S. 2003. Lazy modular upgrades in persistent object stores. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Anaheim, CA. (Oct.). ACM, New York. 403–417.

Buck, B. and Hollingsworth, J. K. 2000. An API for runtime code patching. *J. High Perf. Comput. Appl. 14*, 4, 317–329.

Dmitriev, M. 2001. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*. Portland, OR. (Oct.). 14–18.

Duggan, D. 2001. Type-based hot swapping of running modules. In *Proceedings of the International Conference on Functional Programming*. 62–73.

Frieder, O. and Segal, M. E. 1991. On dynamically updating a computer program: From concept to prototype. *J. Syst. Softw. 14*, 2 (Sept.), 111–128.

Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. New Orleans, LA. (Feb.). 87–100.

Garlan, D., Krueger, C. W., and Staudt, B. J. 1986. A structural approach to the maintenance of structure-oriented environments. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. (Palo Alto, CA). ACM, New York, 160–170.

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

Grossman, D. and Morrisett, G. 2000. Scalable certification for Typed Assembly Language. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*. Montreal, Canada. R. Harper, Ed. Lecture Notes in Computer Science, vol. 2071. Springer-Verlag, New York.117–146.

Gupta, D. 1994. On-line software version change. Ph.D. dissertation, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur.

Gupta, D. and Jalote, P. 1993. On-line software version change using state transfer between processes. *Softw.—Pract. Exp. 23*, 9 (Sept.), 949–964.

GUPTA, D., JALOTE, P., AND BARUA, G. 1996. A formal framework for online software version change. *Trans. Softw. Eng. 22*, 2 (Feb.), 120–131.

HICKS, M. 2001. Dynamic software updating. Ph.D. dissertation, Department of Computer and Information Science, University of Pennsylvania.

HICKS, M., MOORE, J. T., AND NETTLES, S. 2001. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, UT. ACM, New York, 13–23.

HICKS, M., WEIRICH, S., AND CRARY, K. 2000. Safe and flexible dynamic linking of native code. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*. Montreal, Canada. R. Harper, Ed. Lecture Notes in Computer Science, vol. 2071. Springer-Verlag, New York. 147–176.

HJÁLMTÝSSON, G. AND GRAY, R. 1998. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*. New Orelands, LA. 65–76.

HOFMEISTER, C. 1993. Dynamic reconfiguration. Ph.D. dissertation, Computer Science Department, University of Maryland, College Park, MD.

JAVA 2002. The Java HotSpot virtual machine, v1.4.1, d2. Available at `http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.%1/JHS_141_WP_d2a.pdf`.

KRAMER, J. AND MAGEE, J. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng. 16*, 11, 1293–1306.

LEE, I. 1983. DYMOS: A dynamic modification system. Ph.D. thesis, Department of Computer Science, University of Wisconsin, Madison.

MAGEE, J., KRAMER, J., AND SLOMAN, M. 1989. Constructing distributed systems in Conic. *IEEE Trans. Softw. Eng. 15*, 6 (June), 663–675.

MALABARBA, S., PANDEY, R., GRAGG, J., BARR, E., AND BARNES, J. F. 2000. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*. Sophia Antipolis, France. In Lecture Notes in Computer Science, vol. 1850.

MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. 1999a. TALx86: A realistic typed assembly language. In *Proceedings of the 2nd Workshop on Compiler Support for System Software.* Atlanta, GA. 25–35.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999b. From System F to typed assembly language. *ACM Trans. Prog. Lang. Syst. 21*, 3 (May), 527–568.

MOSBERGER, D. AND JIN, T. 1998. HTTPERF: A tool for measuring web server performance. In *Proceedings of the 1st Workshop on Internet Server Performance*. Madison, WI. ACM, New York, 59–67.

NECULA, G. 1997. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*. Paris, France. ACM, New York, 106–119.

OEHLER, M. AND GLENN, R. 1997. HMAC-MD5 IP authentication with replay prevention. Internet RFC 2085.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable webserver. In *Proceedings of the USENIX Annual Technical Conference* (Monterey, CA). 106–119.

PESCOVITZ, D. 2000. Monsters in a box. *Wired 8*, 12, 341–347.

PETERSON, J., HUDAK, P., AND LING, G. S. 1997. Principled dynamic code improvement. Tech. Rep. YALEU/DCS/RR-1135, Department of Computer Science, Yale University. July.

PRATT, J. W. AND GIBBONS, J. D. 1981. *Concepts of Nonparametric Theory*. Springer-Verlag, New York.

RAMKUMAR, B. AND STRUMPEN, V. 1997. Portable checkpointing for heterogenous architectures. In *Proceedings of the Symposium on Fault-Tolerant Computing*. Seattle, WA. 58–67.

SEGAL, M. E. AND FRIEDER, O. 1993. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw. 10*, 2, 53–65.

SEWELL, P. 2001. Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages* (London, England). ACM, New York. 236–247.

SOULES, C., APPAVOO, J., HUI, K., WISNIEWSKI, R. W., SILVA, D. D., GANGER, G. R., KRIEGER, O., STUMM, M., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., AND XENIDIS, J. 2003. System support for online reconfiguration. In *Proceedings of the USENIX.* San Antonio, TX. 141–154.

TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. 1997. A survey of active network research. *IEEE Commun. Mag. 35*, 1 (Jan.), 80–86.

TOOL INTERFACE STANDARDS COMMITTEE. 1995. Executable and Linking Format (ELF) specification.

TRIDGELL, A. AND MACKERRAS, P. 1996. The rsync algorithm. Tech. Rep. TR-CS-96-05, Canberra 0200 ACT, Australia. `http://samba.anu.edu.au/rsync/`.

WEBSTONE. 2001. Mindcraft—WebStone benchmark information. `http://www.mindcraft.com/webstone`.

Received November 2003; revised May 2004; accepted September 2004