Dynamic Software Updating Using a Relaxed Consistency Model

Haibo Chen, *Member*, *IEEE*, Jie Yu, *Student Member*, *IEEE*, Chengqun Hang, Binyu Zang, and Pen-Chung Yew, *Fellow*, *IEEE*

Abstract—Software is inevitably subject to changes. There are patches and upgrades that close vulnerabilities, fix bugs, and evolve software with new features. Unfortunately, most traditional dynamic software updating approaches suffer some level of limitations; few of them can update multithreaded applications when involving data structure changes, while some of them lose binary compatibility or incur nonnegligible performance overhead. This paper presents POLUS, a software maintenance tool capable of iteratively evolving running unmodified multithreaded software into newer versions, yet with very low performance overhead. The main idea in POLUS is a relaxed consistency model that permits the concurrent activity of the old and new code. POLUS borrows the idea of cache-coherence protocol in computer architecture and uses a "bidirectional write-through" synchronization protocol to ensure system consistency. To demonstrate the applicability of POLUS, we report our experience in using POLUS to dynamically update three prevalent server applications: vsftpd, sshd, and Apache HTTP server. Performance measurements show that POLUS incurs negligible runtime overhead on the three applications—a less than 1 percent performance degradation (but 5 percent for one case). The time to apply an update is also minimal.

Index Terms—Maintainability, reliability, runtime environments.

1 Introduction

1.1 Motivation

The scale of software has increased dramatically in the past two decades, as have the bugs and security vulnerabilities. Despite progress made in software engineering with better programming models, improved developing methods, and more effective testing tools, it is undeniable that software is still far from perfect, and this trend is likely to continue. More importantly, software is often required to adopt changes that add new functionalities to support business needs. A previous empirical study [1] on software evolution shows that the number of functions in OpenSSH has increased more than three times within five years. Consequently, there has been an increasing number of software updates to fix bugs, close vulnerabilities, and evolve new features.

Static updating is the traditional approach to evolve software into newer versions; it involves stopping the running software, applying the updates, and restarting the software again. For example, the Windows Update system [2] upgrades current software by downloading the newer versions of packages (e.g., DLLs) and then reboots the software or even the operating system to apply the updates. Such a stop-and-restart approach inevitably disrupts the execution of running services, thus decreasing the availability of software. One previous study [3] indicated that 75 percent of about 6,000 outages in highly available applications were caused by hardware and software maintenance. Since such service disruptions are ill affordable for many mission-critical systems, such as air control systems, credit card authorization, and brokerage operations [4], these systems demand highly dependable services and require services to be available 24X7.

Dynamic updating [5], [6], or live updating, is a promising software maintenance technique aiming to increase software dependability. It is much cheaper and less complex compared to hardware-based approaches such as hot/cold standby [7], [8]. It also avoids service disruption by allowing the code and data of the running systems to be directly updated on the fly. It thus allows dynamically fixing bugs, closing security vulnerabilities, and upgrading software with new features without requiring service downtime. Besides, it could also enable online program profiling and analysis, as well as software testing and debugging. Hence, such a technique has gained considerable interest and popularity with both researchers and practitioners.

1.2 Limitations of Existing Systems

So far a number of systems have been designed to support dynamically updating of software systems. Examples include Podus [5], OPUS [9], Ginseng [10], and many others. Generally, existing systems can be categorized into two types: 1) restrictive ones that only support limited types of updates that do not modify global state [9], [11], and 2) permissive ones that support flexible changes. Patches in the former ones can usually be applied at various execution

- *H. Chen and B. Zang are with the Parallel Processing Institute, Fudan University, Room 301, Software Building, 825 Zhangheng Road, Shanghai, P.R. China 201203. E-mail: {hbchen, byzang}@fudan.edu.cn.*
- *J. Yu is with the ACAL Lab, CSE Department, University of Michigan, 2260 Hayward Street, Ann Arbor, MI 48109-2121. E-mail: jieyu@umich.edu.*
- *C. Hang is with the Microsoft (China) Ltd., 8th Floor, Grand Gateway II, 3# HongQiao Road, Xu Hui District, Shanghai, P.R. China 200030. E-mail: chhang@microsoft.com.*
- *P.-C. Yew is with the Department of Computer Science and Engineering, University of Minnesota at Twin Cities, 4-192 EE/CS Building, 200 Union Street, SE, Minneapolis, MN 55455. E-mail: yew@cs.umn.edu.*

Manuscript received 5 Aug. 2008; revised 8 Feb. 2009; accepted 30 Nov. 2009; published online 6 Aug. 2010.
Recommended for acceptance by E. Di Nitto.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-08-0242.
Digital Object Identifier no. 10.1109/TSE.2010.79.

points. In contrast, patches in the latter ones should usually be applied at specific points, which are generally referred to as *update-points*, in which the code and data to be updated are not being executed or referenced.

While previous systems can support flexible changes to running software, there are limitations that restrict their wide adoption in some scenarios. First, though multicore hardware has been commercially available for years, there are currently few dynamic updating systems that can support updates to broadly use multithreaded application software when changes involve data, due to the ubiquitous yet complicated interactions among threads. Second, as the points where an update can be safely applied is usually hard to find under their (usually strict) restrictions, previous systems may suffer from the constraint of update timing, which would delay some critical updates (e.g., security patches). Third, many of them require extensive changes (e.g., compiler transformation) to the code structures and data layouts of existing software [10] to provide such software with the dynamic updating capability, thus losing the backward binary compatibility and incurring a certain level of performance overhead. Fourth, most existing dynamic updating systems inadvertently assume the integrity of the targeted software, without any precaution that the targeted software might have already entered a tainted state, such as a deadlock. Finally, some of them incur nontrivial performance overhead to normal software execution, preventing their adoption in performance-sensitive environments.

1.3 Our Contributions

This paper presents POLUS, a *PO*werful *L*ive *U*pdating *S*ystem for *existing* software. POLUS introduces a relaxed consistency model to support *permissive* changes involving both code and data to *multithreaded* application software without the *update timing constraint* in previous systems, yet retains *backward binary compatibility* and incurs very low performance overhead. Further, POLUS is also built with effective mechanisms to roll back already committed updates and to fix already tainted states for running software.

To demonstrate the applicability of POLUS, we have implemented a prototype system and evaluated POLUS using three prevalent sever applications that demand nonstop features: *vsftpd* (a commonly used FTP daemon), *sshd* (secure shell daemon) in OpenSSH suite, and *Apache* HTTP server (httpd). *vsftpd* and *sshd* are single-threaded and *httpd* is with multithreading enabled. All updates to these applications are generated from realistic software releases over a relatively long period—from version 2.0.0 through 2.0.4 for vsftpd, 3.2.3p1 to 3.6p1 for sshd, and 2.1.7 to 2.2.0 for httpd. Although a complete automation of patch code generation is impossible, we have developed a source-to-source compiler to automatically generate most parts of the patch code for dynamic update. Performance measurements show that POLUS only incurs a less than 1 percent performance degradation (but 5 percent in one case) for the connection time and transfer rate of the three systems. Also, the time to completely evolve an application into a newer version is minimal (less than 70 ms for all tested cases).

In short, the contributions of this paper are as follows:

1. A relaxed consistency model for dynamic software updating and a "bidirectional write-through" synchronization protocol to maintain system consistency.
2. The design and implementation of a powerful dynamic software updating system that supports dynamic updates to multithreaded applications, yet with backward compatibility and very low performance overhead.
3. The demonstration that POLUS can deliver realistic updates to real, large, and complex server software without disrupting its service. Our experience shows that dynamic software updating is a promising approach to evolve contemporary complex software.

The next section provides a brief overview of the system architecture and the approach taken by POLUS. Then, we introduce POLUS patches and the process of patch generation in Section 3. Next, we show how to apply POLUS patches in Section 4. In addition, we provide several case studies in using POLUS to update real-life server applications in Section 5, and provide our performance results in Section 6. Section 7 presents a discussion on related work. We close this paper with a discussion on further work and a conclusion.

This paper is an expansion of an earlier version published in the *Proceedings of the 2007 International Conference on Software Engineering* [12]. Compared to the conference version, this paper models the proposed approach as a relaxed consistency model in dynamic updating, which is supported by a bidirectional synchronization protocol. It also adds a running example through the paper to illustrate the system and an approach to give a more clear presentation. Besides, this paper argues on the safety issues based on the proposed relaxed consistency model. Finally, this paper expands the description of the design, implementation, evaluation, and work related to our system.

2 POLUS: Approach and Overview

In this section, we present an overview of the system architecture, describe the relaxed consistency model, and use a running example to illustrate the working flow of POLUS.

2.1 An Overview of POLUS

POLUS supports flexible software updates that add or change[1] types, global variables, and functions. As the deletion of them does not require special handling, POLUS simply keeps them in memory for possible future rollbacks of patches. POLUS applies updates at the function level by treating each function as a black box, thus does not handle updates to local variables. To retain backward compatibility, POLUS uses binary rewriting to replace the prologue of a function with a jump instruction to redirect a function invocation from its old version to a newer version.

To improve the interoperability and support recovery of already tainted state, POLUS provides several callbacks in dynamic patches so that software vendors can easily provide their analysis and recovery code in the patch to avoid such situations. There are also some tools in the literature that are able to analyze the state of running software and repair it if corrupted [13]. Such tools can also be integrated into POLUS.

1. A type, variable or function is considered as added if it is newly introduced (e.g., with new names or signatures) in a new version, as changed if the old version contains it but the representation is changed in the new version, as deleted if the new version never uses it again.

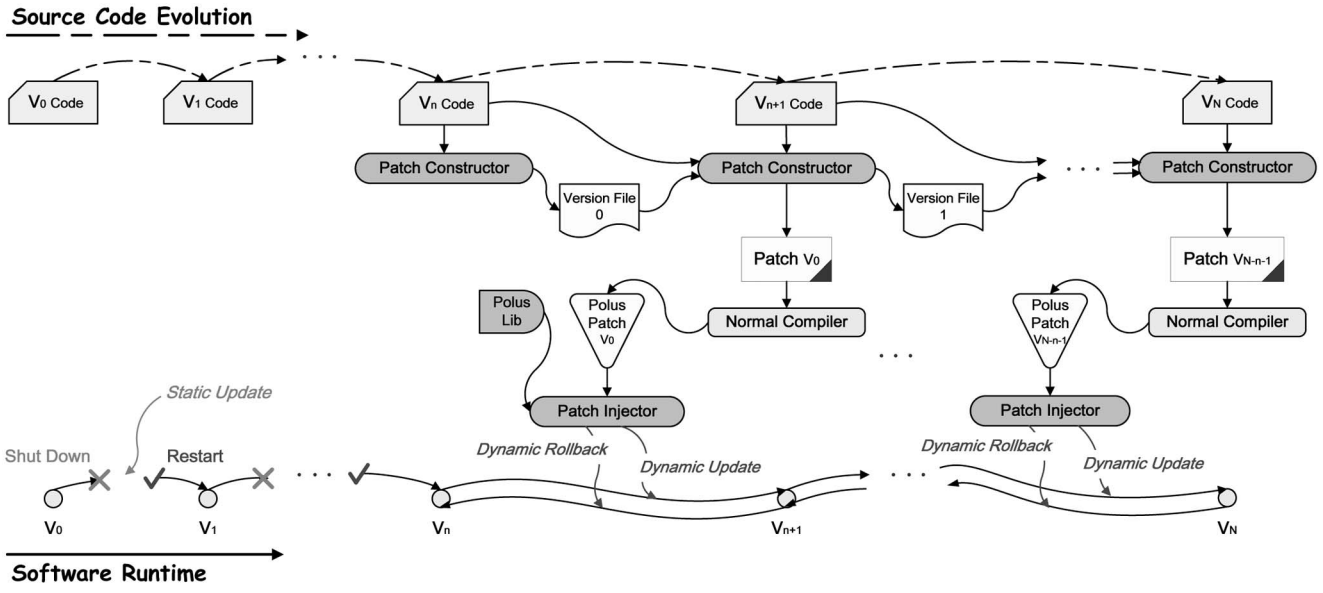


Fig. 1. An overview of POLUS and its working flow.

Fig. 1 gives an overview of POLUS, which is composed of three components: *a patch constructor*, in the form of a source to source compiler which detects the functionality differences between two successive software versions and generates the POLUS patch files, *a patch injector*, which is a running process that applies the updates, and *a runtime library* which provides some utility functions to manage POLUS patches for the patch injector.

Fig. 1 also shows the life cycle of software and general working flow of dynamic updating using POLUS. Traditional ways of software evolution involve stopping the running software, applying the updates, and restarting the software again, while dynamic updating supports changes to code and data on the fly. To retain binary compatibility, a dynamic update to the software can be started in any running version. A POLUS patch is obtained by analyzing the differences of two successive software versions. To facilitate iterative updates, a version file is used to control the renaming of functions and data in the patches. The patch is then compiled using regular compilers to generate a dynamic patch as a shared library. The POLUS runtime library will be injected into the running software before the first update. The patch injector will inject the dynamic patch to the running program, facilitated by the POLUS runtime library.

2.2 Relaxed Consistency Model in POLUS

The core in the relaxed consistency model is to allow the concurrent activity of both the old and new code and the coexistence of both the old and new representation of a data structure. However, the old (new) code is only allowed to operate on the old (new) data, respectively.

To ensure system consistency, POLUS borrows the idea of cache coherence protocol in computer architecture and supports an update-based coherence protocol; update to either version of data will be transactionally propagated to the other version of data using the provided state synchronization function [14]. We call this protocol the "bidirectional write-through" synchronization protocol, as a contrast to the *one way and one time* state synchronization protocol in traditional update-point-based systems [6], [10]. Accordingly, POLUS requires the states be bidirectional convertible between the old and the new processes/threads, instead of only one way (i.e., from old to new) in previous systems. Under such a situation, rolling back a committed patch is similar to applying a patch, simply by treating the old version of code/data as the new ones.

Relaxing the consistency requirement of dynamic updating system can naturally overcome the update timing issues with previous systems. The coexistence of both the old and new code and data allows an update to be applied even if the code and data are currently in use. Hence, POLUS naturally suits the prevalent multithreading programming model in which the code and data to be updated are usually referenced by multiple threads and can hardly be quiescent.

To keep track of changes to data, POLUS write-protects either version of the data during the updating process using the debugging APIs provided by operating systems (e.g., *ptrace* in Unix-like operating system and *DebugActiveProcess* in Windows). Such APIs allow a process to gain control over another process, and track a write access to the protected data using the signal mechanism (catching and checking the *SIGSEGV* signal). When there is no function manipulating the old version of data, the update process can be safely terminated.

2.2.1 Ensuring System Consistency

POLUS shares a similar assumption with previous approaches; new processes/threads running new code can be executed upon the states changed by the old processes/threads running old code and the state is convertible from the old to the new version of software. The difference is that POLUS requires the state conversion to be *bidirectional* instead of one way. Thus, POLUS additionally requires programmers to provide a state synchronization function that converts the state from the new version back to the old version. For some rare cases that the old and new version of

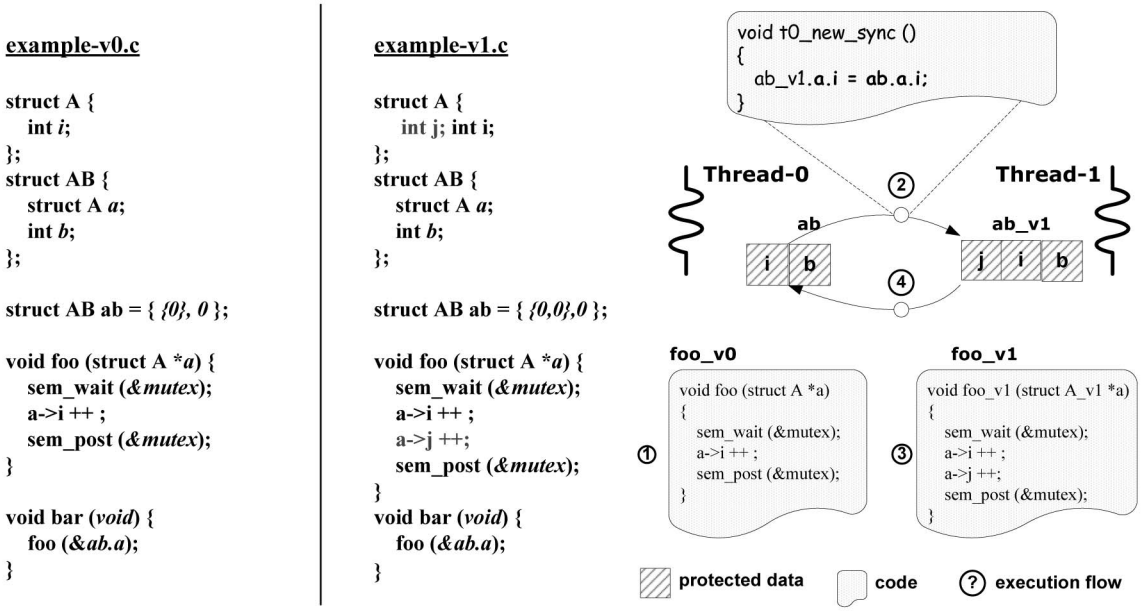


Fig. 2. An example software evolution (left) and how POLUS handles it (right).

code that has different programmer-designated rules on some data structures, writing such synchronization functions would require a deep understanding of the rules. Fortunately, in most cases (such as the tests we have conducted), the state synchronization functions for both directions are similar.

POLUS needs to ensure that at no time will a thread (i.e., code) execute upon a state with a different version from that of the thread (i.e., code). This property ensures that, given the premise that the states are bidirectional convertible, running new threads while the old threads are still active will not cause state inconsistency if the old or new threads are executed with proper version of states. The state synchronization protocol, which acts like a *state proxy* in POLUS, creates the illusion that an old (new) thread only interacts with other old (new) threads, respectively. Thus, both the old and new threads can see a consistent system state (though different versions). When an old thread has switched to execute the new version of code, the thread will transparently become a new thread as it now executes upon the state of a new version. Hence, when all old threads become new threads, the system state will evolve into the newer version, without loss of system state.

2.3 An Example Software Update

Fig. 2 shows an example software evolution that changes the type representation by adding an additional member (i.e., j) to structure A, and how POLUS handles the update. It is possible that there are still threads (e.g., thread-0) executing in *foo* but have not increased *a->i* yet. If a trivial update scheme simply applies update at this point, the increment to *ab.a.i* by thread-0 might be lost, which breaks the system integrity. POLUS handles this by write-protecting both versions of data (i.e., *ab* and *ab_v1*). During initializing the patch, the POLUS runtime will first initialize the value of *ab_v1* from the current state by invoking *t0_new_sync*. When thread-0 tries to modify *ab* (step 1), a signal will be raised to the POLUS runtime, which will invoke the state synchronization function (e.g., *t0_new_sync*) to propagate the change to *ab_v1* (step 2). Similar procedures will take place when thread-1 tries to modify *ab_v1* (steps 3 and 4).

3 POLUS Patches

Unlike static patches that only reflect literal changes, POLUS patches describe the functional changes between two versions of a program. This section describes the essential elements, correctness issues, and automatic generation of POLUS patches.

3.1 Essential Elements in a Patch

A patch in POLUS is usually in the form of a full-program patch, which describes the flexible changes between two versions of a program. According to Neamtiu et al. [1] and Stoyel et al. [15], most software updates are composed of updates to type definitions, functions, and global variables. Therefore, a POLUS patch, at a minimum, should be able to express such changes. Moreover, POLUS patches need to describe ways to maintain state consistency during an update. Finally, to provide interoperability to programmers, POLUS patches also include a set of optional callbacks to execute programmers' own code during the process of dynamic update.

Fig. 3 shows a simplified version of the POLUS patch corresponding to the example in Fig. 2. In the followings, we use it as an example to illustrate the essential components that form a valid POLUS patch, as well as the way to generate them.

3.1.1 Type Changes

Types are not concrete objects in a program, but only provide information for compilers. Therefore, for added types, the type definitions are included to the patch to make

example.polus.c

```
/* external definitions */
...
/* renamed types and variables */
struct A_v1 {
   int j;  int i;
};

struct AB_v1 {
   struct A_v1 a;
   int b;
};
struct AB_v1 ab__v1 = { {0,0}, 0
/* added and renamed functions */
void foo_v1 (struct A_v1 *a) {
    sem_wait (&mutex);
    a->i ++ ;
    a->j ++;
    sem_post (&mutex);
}
void bar_v1 (void) {
   foo_v1 (&ab_v1.a);
}
/* data synchronization function */
void old_ab_post_sync (void) {
   ab.a.i = ab_v1.a.i;
}
void new_ab_post_sync (void) {
   ab_v1.a.i = ab.a.i;
}
static void register_patch (void) {
    /* register functions to be updated */
   ofIdx0 = polus_function(IS_OLD, bar, "bar", ...);
   nfIdx0 = polus_function(IS_NEW, bar_v1,
"bar_v1", ...);

   /* register data to be updated */
   odIdx0 = polus_data(IS_OLD, &ab, sizeof(ab),
                       old_ab_post_sync, ...);
   ndIdx0 = polus_data(IS_NEW, &ab_v1, sizeof(ab_v1),
                       new_ab_post_sync, ...);
   /* mapping between old and new functions */
   polus_function_mapping(ofIdx0, nfIdx0);

   /* mapping between old and new data */
   polus_data_mapping(odIdx0, ndIdx0);

   /* mapping between function and data */
   polus_f2d_mapping(ofIdx0, odIdx0);
   polus_f2d_mapping(nfIdx0, ndIdx0);

   /* mapping between data and function */
   polus_d2f_mapping(odIdx0, ofIdx0);
   polus_d2f_mapping(ndIdx0, nfIdx0);
}
```

Fig. 3. A simplified POLUS patch corresponding to the example evolution shown in Fig. 2.

the compiler be aware of the new types. For changed types, since the old and new types share the same name in the program, the new type definition should be renamed (by appending a version number). Further, types are considered as changed in a recursive manner; if the types of any members in a type are changed, this type is also considered as changed. In the above example, as *A* is changed and *AB* contains *A*, thus both *A* and *AB* are considered as changed, and thus renamed with a corresponding version number (e.g., A_v1 and AB_v1).

3.1.2 Global Variable Changes

A global variable is considered as changed if its type (thus the actual storage class and size) is changed. For changed variables, the POLUS patch should include code that registers both the old and the new global variables and their mappings. Such code will be invoked by the POLUS runtime when an update is initiated. The registration information will later be used by the patch injector to do data protection and data synchronization. In the above example, as the type *AB* is changed, the variable *ab* is thus changed and renamed with the version number. Calls to the library function *polus_data*, which registers the address, size, and associated state synchronization function, are added in *register_patch* to register both *ab* and *ab_v1*. The call to *polus_data_mapping* is also added to register the mapping between the two versions of data.

For changes to complex global variables, such as arrays and pointers, POLUS first identifies an array or a pointer by parsing the type of a specific variable. Then POLUS uses a recursive tracing process to register them. For pointers, the pointers themselves (both the old and new) and the data they point to should be registered, and be protected later by POLUS runtime for synchronization. For a linked list, each node in the list is supposed to be registered and protected so that the consistency of the update can be assured. Therefore, a recursive search should be performed from the head node to find all nodes that should be protected.

Note that for flexible languages such as C, compilers usually have difficulties in pointer analysis. Hence, POLUS uses a conservative way and raises warnings for pointers with more than one aliased instance. In the absence of aliases, POLUS can automatically generate the tracing code.

3.1.3 Function Changes

Other than literal changes, a function should also be considered as changed if it uses a variable whose type has been changed. In the above example, *bar* is changed since it refers to the global variable *ab*. If the types or global variables used by the function are changed, they should be renamed in the new function accordingly. Besides literally new functions, a function is also considered as added if its signature (e.g., argument list or return type) is changed [6], to prevent possible type errors after update.

In the above example, the argument type of *foo* in the new version is changed to take a pointer to *struct A_v1* and *bar* is also changed accordingly to call *bar* with the address of *ab_v1.a*. If we simply treat *bar* as a changed function and the update occurs just before old *bar* calls the old *foo*, new *foo* will expect a pointer to *struct A_v1* but the old *bar* has only provided a pointer to *struct A*. Thus, a type error occurs. Hence, POLUS treats the new version of *foo* as an added function. In this way, old *bar* will continue to call old *foo* after the update, and new *foo* will be called by new *bar* only when *bar* is invoked after its own update.

Similarly to global variables, the old and new functions should also be registered using the POLUS library function (e.g., polus_function). In the above example, function *bar* is changed and *foo_v1* is an added function.

3.1.4 Function and Data Mappings

To assist POLUS runtime to perform a consistent update, a POLUS patch needs the mapping information between the old and the new functions, between the old and the new data, and between functions and data. Hence, a set of API is provided in POLUS library to assist the registration of the mapping information. For instance, if a function is changed, the mapping between the old and the new functions should be registered with both identifiers of the old and the new functions. POLUS runtime will use the mapping information to insert an indirection in the old function. Further, to determine whether an update is completed, the patch injector needs to know what data a function uses and what functions use a specific variable. Hence, POLUS patches should contain code describing the mapping information using the API in POLUS library. Fortunately, most of the mapping information can be inferred automatically by POLUS patch generator, which will be illustrated later. In the above example, the automatically generated calls to *polus_f2d_mapping* and *polus_d2f_mapping* register such information.

3.1.5 State Synchronization Functions

There are two synchronization functions for each pair of data to be updated, namely, *old_sync* and *new_sync*, which are called after modifying the old and new data accordingly. Besides, POLUS allows the synchronization functions to be called before and after the associated data is modified (i.e., pre_sync_cbfn and post_sync_cbfn). Normally, POLUS uses the *post_sync_cbfn*. But for some complex data structures,

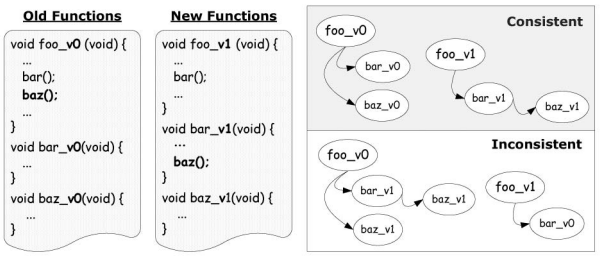


Fig. 4. An example of a possible version inconsistency problem of dynamic updating.

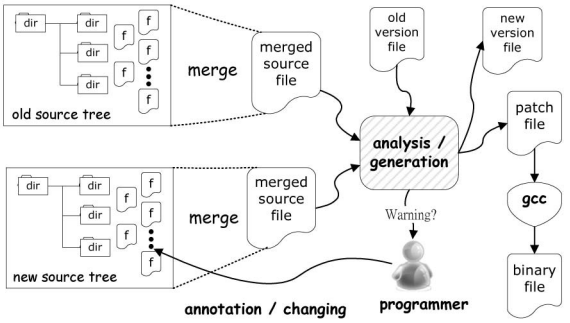


Fig. 5. The process of generating a POLUS patch.

POLUS uses the pre_sync_cbfn to record the old value before modifying the data. For example, we might need to record what the node in a linked list previously points to in order to synchronize the old and new linked lists.

3.1.6 Update Callbacks

To provide interoperability and flexibility, POLUS allows users to execute their own code during the update process in the form of callback functions. For example, these callback functions can be used to recover from an already tainted state in an application. Users can define several callbacks in a patch, according to the time at which they are invoked. More details about update callbacks will be described in Section 4.1.4.

3.2 Ensuring Patch Correctness

The consistency of a system state during an update depends on the correctness of the state synchronization functions and the version consistency of function calls. For most simple cases, such as the example shown above, an analyzing compiler can automatically generate the correct synchronization functions and ensure version consistency. However, for complex changes that reflect programmer-designated rules (e.g., a should always be equal to b), POLUS might need programmers' involvement to ensure patch correctness. In addition, special treatments are required to handle nonexit functions.

3.2.1 Dynamically Allocated Data

If the old data of a patch are statically allocated, then the new data should also be allocated at compile time accordingly. Otherwise, the new data should be dynamically allocated. For example, if the type of the node in a linked list is changed, POLUS should allocate a new list and keep the old and the new in sync. Since POLUS cannot statically obtain the length of the list, it is impossible to allocate the new list at compile time. Our solution is to allocate it at runtime via update callbacks. In addition, according to our experience, for dynamically allocated data, POLUS must check its existence prior to the update to decide whether to allocate a new data or not. For example, if a global pointer P of type A is defined with an initial value NULL, and the old data with type A is allocated at runtime. In this case, POLUS must check whether P is NULL or not before update to decide whether to allocate new data or not during update.

3.2.2 Version Consistency

Updates in POLUS are at the granularity of function calls. As a result, old functions might interact with new functions and cause inconsistency. For the example in Fig. 4, function *foo* is defined to call *bar* and *baz* in turn. Then, a change moves the function call to *baz* from *foo* to *bar*. Suppose the update occurs prior to the original pair of calls. Both calls to *bar* and to *baz* will be redirected to the new versions of them. As a result, *baz* will be called twice, potentially leading to an error. Therefore, for this case, POLUS should treat the new version of *bar* and *baz* as added functions. More precisely, old *foo* will continue to call old *bar* and old *baz* after the update. We call such a problem the *version consistency problem*, as there are functions that should be updated together (e.g., *bar* and *baz*). This problem is usually addressed by specific handling. In recent literature, automatic method has been developed [16]. We believe their method is useful and can be integrated with our system.

3.2.3 Main Function and Infinite Loop

In POLUS, active code will proceed with the old version after the update, and the new code takes effect in the subsequent calls. However, there are some functions that never exit. Thus, their new versions will never take effect. Main function is a typical example. In many event-driven servers, infinite loop is another example. There are no general solutions to this problem. However, for some special cases, we can solve it by some tricks. One possible solution is using stub functions [6]. Updating the main function of sshd shown in Section 5.1.1 is such an example. For infinite loops, we have implemented *loop extraction* [10], which extracts the body of a loop as a separate function to be updated, as an optional solution at the cost of some binary compatibility. Fortunately, in our current evaluation on software evolution, we did not encounter the infinite loop problem.

3.3 Patch Generation

To relieve programmers of the tedious work in writing patches, we have implemented a *patch constructor*, which is a source-to-source compiler based on CIL-1.3.6 [17] and Ginseng-1.2.1 [10] using OCaml. Fig. 5 shows the process of patch generation in POLUS.

The first step of the patch generation is the merge process. Modern software is usually composed of multiple files. We must ensure that if one function or type is changed, all affected code and data must be updated accordingly. For simplicity, POLUS merges all related files into a single file using the merge feature in CIL [17]. In the merge process, POLUS carefully handles the naming conflict by associating clashed names with their filename.

Then, the patch generator generates code to notify the patch injector to resolve such name conflicts by scanning the original binary file.

Next, the patch generator finds changed types, global variables, and modified functions by comparing the syntax tree of both the old and the new versions of files. For each changed type, the patch generator finds all global variables that derive from the type and adds them to the changed variables list. Then, for each changed variable, a state synchronization function is generated to maintain the coherence between the old and the new variables. Also, the patch generator gathers all functions that use each changed variable and registers the mapping relationship between them. For each modified function, the patch generator detects all changed global variables it uses and registers their relationship.

To support iterative patching, the patch generator uses version files [10] to record the patch history of functions, types, and variables to avoid naming conflicts. The patch generator renames each function, type, and variable in the patch file according to its patch history. For example, if function *foo* has been updated three times, then the version number for *foo* in versionnonexist nature data file is 3 and its name in the new patch file is *foo_v4*. Patch generator maintains a global version to record the total update times. The version for each element in the version file may not be the same if the history of individual updates differs.

After the analysis process, the patch generator generates a single file that contains all changed type definitions, all changed global variables, all modified functions, and some support code to register the relationship between variables and functions.

Finally, this patch file is compiled using regular compilers (such as GCC) to generate a dynamic patch file in the form of a shared library.

It is possible that the patch generator may fail to obtain all changed information in the presence of pointer aliases and void pointer casting. To solve this problem, the patch generator will generate warnings that ask operators to adjust the source code or add needed source annotations to make sure that the patches generated are correct.

4 Applying POLUS Patches

This section describes the runtime support in POLUS to apply dynamic patches on the fly. We will first describe the necessary support for dynamic updates, followed by an overview of the process to apply a POLUS patch.

4.1 Support for Dynamic Updating

POLUS provides a *runtime library* consisting of a number of utility functions to maintain the update information of each software update, as well as a *patch injector* which applies a patch. In the following, we will describe the basic mechanisms to support the dynamic updating of POLUS patches.

4.1.1 Function Indirection

POLUS uses binary rewriting to implement the update to functions. To implement function indirection, POLUS inserts an indirect *jump* instruction in the prologue of the original function to force all function calls from the old function to the new function. Before doing this, POLUS first saves the original code, and then checks the *program counter* to ensure that the program is not executing the code to be replaced. If the check fails, POLUS aborts the current update process and performs a retry. However, this rarely happens in practice.

As POLUS supports iterative updates to a single function, it is carefully designed to avoid multiple indirections, which can degrade performance. New functions are permitted to directly call other new functions without indirection. POLUS keeps all the versions of functions in memory for future rollbacks, which incurs some memory overhead. However, as one of our goals is to support rollbacks of committed patches, keeping them in memory will make rolling back and updating forward easier.

4.1.2 State Management

In POLUS, both the old and the new instances of data are allowed to coexist simultaneously. As old functions manipulating the old instances may still be active, there might be concurrent accesses to the old or the new instances. To avoid inconsistency, POLUS needs to track the accesses to either version of data and uses state synchronization functions to maintain system consistency.

When a dynamic update is being applied, the patch injector write-protects (e.g., *mprotect* in Linux) both the old and the new versions of an instance and associates a signal handler (e.g., *SIGSEGV* handler in Linux) to catch each write attempt to either version of the instance. The signal handler will invoke the corresponding state synchronization function to transfer the modified state from one version to the other. To be more flexible, POLUS allows state synchronization functions to be called either before or after (or both) the commitment of a write access. This page level protection might cause false sharing, that is, some writable data that are not involved in the update will also be write-protected. On receiving the signal, POLUS first retrieves the fault address, and judges whether this fault occurs due to the false sharing through the read-write bits recorded and the fault address retrieved. The only difference for a false sharing is that POLUS need not invoke the state synchronization function to transfer the state from/to the new/old versions of data.

In our experience, some of the changed global variables may be read-only throughout their whole life cycle. For such changes, there is no need to write-protect any instance of data and maintain their consistency. As the patch constructor ensures the old (new) instances will only be used by the old (new) functions, the old instances will not be used when all old functions accessing them have completed.

4.1.3 Stack Inspection

POLUS needs to track the in-flight call stack of each thread to determine if a function is still active. To obtain the runtime call stack, POLUS inspects the call stack of each thread. In Linux on x86, it is achieved by iteratively scanning the frame pointer (*percent ebp*) and return address (above the frame pointer) in the call stack of each thread. Usually, this method is viable. However, some leaf functions do not have stack frames in practice (e.g., *select* in libc), which makes it difficult to obtain the return

addresses for these functions, leading to an incorrect result [18]. There are two possible solutions. The first is to analyze the prologue of the current function and obtain the return address through the analysis result rather than the frame pointer, which is used by gdb. The other one is to scan the call stack word by word between the addresses pointed to by the current stack pointer and the current frame pointer for the return address, which is similar to Linux kernel dump. The first one is more powerful and robust, but it is quite complicated to implement. The second one might be incorrect under some circumstances, since the content of the stack could mislead the scanning process. In POLUS, since it is only required to check those functions to be updated, the possibility of an incorrect scanning is very low. In practice, we choose the second solution for simplicity, and it works well for all the applications we evaluate.

After the runtime call stack is obtained, POLUS will track it dynamically. POLUS replaces the return address of each old function to a supplied stub function. The stub function will remove the calling thread from its active thread list, and return to the correct function address. Therefore, POLUS is able to find out whether a function is in the current stack of all threads at anytime during the update process.

4.1.4 Update Callbacks

POLUS provides several opportunities during an update for programmers to insert their own code. The code will be used to allocate new data, to analyze the current state of a running program, or to recover from an already tainted state at runtime. We refer to the code as *update callbacks*. Currently, there are five kinds of update callbacks in POLUS according to the time; they are called:

1. ***preupdate callbacks***, that are called before an update process is started.
2. ***thread callbacks***, that are invoked each time a thread leaves a function being updated.
3. ***function callbacks***, that are called when all threads have left a function being updated.
4. ***data callbacks***, that are invoked when all threads using a data structure have returned from the functions that manipulate the instance of the data structure.
5. ***postupdate callbacks***, that are called when an update process is to be terminated.

The following situations can benefit from these callbacks:

Dynamic new data allocation. As previously mentioned, for the old data that are dynamically allocated, it is impossible to allocate the space for the corresponding new data at compile time. A *preupdate callback* is an ideal mechanism to execute the allocation code. Updating a dynamically allocated linked list is a typical example. The new list can be allocated in the *preupdate callback* function, which will traverse the old list and allocate the new nodes accordingly.

Recovery from tainted states. Existing approaches assume the state for of the running software being correct when an update is being applied. However, it is likely that the running software is buggy and may have entered a tainted state (such as a deadlock situation). For example, a known vulnerability on SSL connection in Apache 2.0 will cause a child process to enter an infinite loop, risking denial of service.[2] As a considerable number of software updates are to fix existing bugs, we feel it is necessary to consider the detecting and fixing of buggy situations during update.

To support recovery from a tainted state, patch vendors can selectively provide their checking code in update callbacks to detect possible buggy situation and fix them if needed. There are many algorithms to perform such analysis and recovery [13]. The POLUS framework just provides mechanisms for these algorithms to be invoked when needed. To handle the case in Apache 2.0, one can provide code in the *preupdate callbacks* to check for the infinite loop and break it if necessary. However, not all buggy situations can be easily resolved. For example, in some memory-leaking programs, it will be hard to reclaim all leaked memory if it cannot trace all of them.

4.1.5 Hijacking Running Processes

One key issue during our update process is to hijack the running process to be patched (RPP). To apply an update, we need to load the *POLUS runtime library* and *POLUS patches* to RPP's address space first. However, no process is able to load these two items to RPP's address space at runtime but RPP itself. Therefore, we use some tricks to do this task.

We first create a code playground [9] in RPP. More precisely, POLUS maps a range of addresses using *mmap* in RPP, injects code containing *dlopen*, and uses *ptrace* to force RPP to execute the injected code. To regain control after the execution leaves the playground, POLUS appends an "*int3*" to every code in the *code playground* and hijacks the *SIGTRAP* signal.

4.2 Applying a Patch

There are three phases in applying a new patch, as shown in Fig. 6:

- *Patch Initialization:* The patch injector first does some initialization work, such as loading the patch into memory, resolving symbols, and registering the patch information (step 1). Then, it invokes the *preupdate* callbacks in the patch (step 2) and uses stack inspection [9] to get the in-flight call stack of each thread (step 3). For a changed variable, if no thread is executing in any function that references that variable, then updates to these functions could be simply done by function indirection (step 4) and no tracing work is required. Next, the patch injector writes protect all changed global variables currently in use (step 5). Note that the patch initialization work is done when the process to be patched is suspended and all operations are done in atomic.
- *State Synchronization:* The patch injector resumes the execution of the running software. There may be old functions that are still active. They may need to read and write old global variables. To ensure correct execution, the patch injector tracks any write access by intercepting the *SIGSEGV* signal (step 1). It then synchronizes the states by transforming the state from one to the other (step 2).

2. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0748.

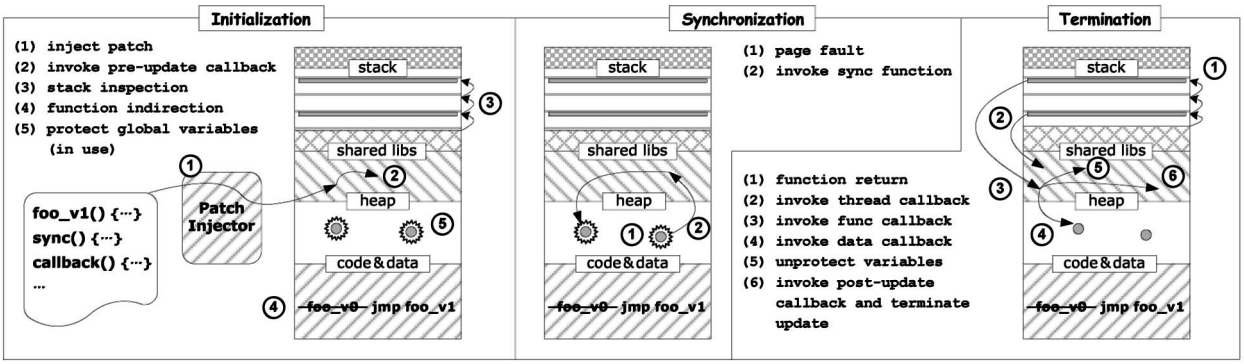


Fig. 6. The working flow of applying a patch.

- *Patch Termination:* The criteria to safely terminate an in-flight update are all threads executing in functions that manipulate changed global variables have become inactive. To determine whether an old function is still active or not, the patch injector maintains a list of active threads for each old function. At the patch initialization stage, the list is initialized according to the in-flight call stacks. The patch injector tracks the thread execution by replacing the return address of the original function with the address of a stub function.

  When an old function returns (step 1), the stub function will remove the executing thread from the thread list, invoke the thread callback (step 2), and return to the caller of the original function. On removing a thread from the thread list, the patch injector checks whether the thread list has become empty or not. If it is empty, the original function is no longer active and the function callback will be invoked (step 3). When all functions manipulating a data structure become inactive, the data callback will be invoked (step 4) and the global variable will be unprotected and marked as unused (step 5). When all changed global variables become inactive, the live update process can be safely terminated and the patch injector will invoke the postupdate callbacks (step 6). The patch injector then performs some cleanup work such as restoring the write-protected memory.

4.2.1 Rolling Back Committed Updates

The process of rolling back a committed patch is similar to that of applying a patch: POLUS treats rollbacks as a special type of updates using existing version of code and data to update the committed one. To roll back a system to a previous version, the state synchronization functions are reused in the rollback process. POLUS uses a flag to indicate whether an update is a rollback or a normal update. To support fast rollbacks, POLUS keeps all old versions of code and data in memory. Although it does incur some resource overhead, doing so allows the running software to freely switch among selected versions. Note that it is possible that the new version contains bugs and the bugs have been triggered. To ensure safety in such a case, one needs to provide their own checking and recovering code in the callback functions to detect and recover the tainted state during a rollback.

5 Case Studies

To demonstrate the applicability of POLUS, we have used POLUS to dynamically evolve three realistic long-running server applications into newer versions, over a period of releases:

1. The Very Secure FTP daemon (*vsftpd*), which is the *de facto* FTP server in UNIX environments. We considered the online evolution from 2.0.0 through 2.0.4.
2. The ssh daemon (*sshd*) from the OpenSSH suite, which is a widely used secure shell daemon. We followed the evolutions from version 3.2.3p1 to 3.6p1.
3. The *Apache* HTTP server (*httpd*), which is a most prevalent HTTP server used nowadays. We tested the upgrades from version 2.1.7 to 2.2.0.

The *vsftpd* and *sshd* are single-threaded software, while *httpd* is usually configured as multithreaded software.

In the following, we first show some real examples and experiences in updating these programs, and then provide the statistical data in updating these three server applications.

5.1 Experience in Updating Three Servers

5.1.1 Handling Main Function

As illustrated previously, updates to main functions are difficult to apply because of their nonexit nature. One major change in main functions is change to initialization code, which will not be executed after the program is already running. Although it appears that updating to such code is useless, it cannot be simply ignored because different initialization code might produce different program states. Therefore, a POLUS patch should provide a proper *preupdate callback* to transfer the old state to the corresponding new one.

Another change is update to the functions directly called by a main function. Fig. 7 shows a code segment in main function of sshd with version 3.5p1. The next version of sshd (version 3.6p1) modified the data structure of *struct monitor*. As a result, the type of the corresponding function *mm_send_keystate* was also changed. As POLUS treats prototype-changed functions as added functions, the newer version of the function can only take effect when its caller is also in a newer version. However, since main function will never exit, the updates to those functions will never take effect. In this example, the newer version of *mm_send_keystate* will never be invoked.

```
int main(int ac, char **av) {
  ...
  for (;;) {
    ...
    ret = select(maxfd+1, fdset, NULL, NULL, NULL);
    ...
    if ((pid = fork()) == 0) {
      /* Child. ... */
      ...
      break;
    }
    /* Parent. Stay in the loop. */
    ...
  }
  ...
  if (use_privsep) {
    mm_send_keystate(pmonitor);
    exit(0);
  }
  ...
}
```

OpenSSH Suite 3.5p1 *sshd.c*

Fig. 7. A Code segment in main function of sshd (version 3.5p1).

One possible solution is using *stub functions*, which invoke the new functions but share the same prototype to the old functions. Thus, the updates can be treated as updates from the old functions to the stub functions. In the above example, POLUS defines a stub function called *mm_send_keystate_stub*, which shares the same prototype with the old version of *mm_send_keystate*. In the body of *mm_send_keystate_stub*, POLUS does corresponding data transformation from the old version to the new version, and then calls the newer version of *mm_send_keystate*.

Using *stub functions*, POLUS can also handle data updates on the main stack between two versions. A concrete example exists in vsftpd (from version 2.0.4 to version 2.0.5), where the type of the data *the_session* on the main stack is changed.

5.1.2 Handling Uninitialized Data

For a change to a global pointer, not only the pointer itself, but also the data it points to should be write protected. However, during updating the three server applications, we found that sometimes the data a pointer points to are uninitialized, which prohibits the protection to the data. For example, when updating sshd from version 3.3p1 to version 3.4p1, we find that the global pointer *channels* sometimes have a NULL value. The problem is that the patch generator does not know the runtime information about the value of the pointer. One possible solution is to check the existence of every pointer in the patch, but this will make a POLUS patch very large and confusing. Another option is to add checking code manually or by program analysis. For simplicity and clarity, we choose the latter one.

5.1.3 Recovery of Tainted State

POLUS is designed to support recovery from a tainted state. In upgrading *sshd*, we found all versions prior to 3.7.1 contain possible buffer management errors.[3] To detect and resolve such a situation, we added checking and recovering code in *preupdate callbacks* in the dynamic patch to check whether the buffer size is valid for each global variable derived from *Buffer* type. Also, we added such code in the *function callback* for each function manipulating such global variables. If the size of a buffer exceeds the defined threshold, the buffer will be truncated in case of a heap overflow.

Although it is sometimes difficult to fix a tainted state and resolving it requires a vulnerability-specific knowledge, we believe our work has raised an important issue to researchers and practitioners about the detection and recovery of tainted states during dynamic updating of running software. Further, as we mentioned previously, our work can be easily integrated with some existing tools (like [13]) to fix a corrupted state.

5.1.4 Update Statistics

Table 1 shows the evolution history of the three applications. We report the total number of changes to functions, types, and global variables from the starting version to the last updated version. Note that POLUS does not really do special handling to the deletion of types, variable, and functions, but simply keeps them untouched for further possible rollback. The number of changes is somewhat larger than other approaches due to the fact that POLUS uses function-level updating. For example, if a type is changed, then all affected functions and variables are affected. Nevertheless, we believe it is justifiable as POLUS retains binary compatibility compared to the compiler-transformation approach [10].

6 Performance Results

A practical dynamic updating system should only cause acceptable performance loss and service disruption. In this section, we report on a set of quantitative performance evaluation aiming to answer the following questions:

- What is the performance overhead incurred by POLUS dynamic updating system in the normal runs of applications?
- How long does it take to evolve a running software application into a newer version and to roll the software back?
- Does POLUS cause significant disruption to the running services when applying an update?
- What is the incurred memory overhead for an update?

The experiments were conducted on a dual Xeon 2.4 GHZ server with 2 GB RAM, a 1 GB Realtek 8169 NIC in 100 M LAN, and a single 73 GB 10k RPM SCSI disk. The systems were configured with a Linux Enterprise edition 4, with kernel version 2.6.9-5.ELsmp. The compiler used is gcc-3.4.3 at optimization level -O2. The reported result is a median of 10 runs.

6.1 Relative Performance

For the three applications, we measured their performance using two metrics—connection time and transfer rate. The test methodologies are borrowed from the Ginseng update system [10], as shown in Table 2. To measure the performance of Apache httpd, we used *ab* (apache benchmark) to issue 50,000 requests for a single 2.4 KB file, with 500 simultaneous threads.

3. http://www.cert.org/advisories/CA-2003-24.html.

TABLE 1
Update Information for Three Applications over Time

Prog.	First version		Last version		Functions			Types			Global variables		
	Ver.	LOC	Ver.	LOC	Add	Del.	Chg.	Add	Del.	Chg.	Add	Del.	Chg.
vsftpd	2.0.0	13,917	2.0.4	14,293	10	4	81	2	0	16	12	1	3
sshd	3.2.3	54,360	3.6	56,960	32	9	512	1	2	6	27	3	11
httpd	2.1.7	319,366	2.2.0	315,381	18	3	195	4	1	5	12	3	3

TABLE 2
Test Methodologies for the Three Applications

Apps	connection time	transfer rate
vsftpd	average time of requesting 1,000 empty files using *wget*	download rate of a single 222MB file
sshd	total elapsed time of 1,000 requests divided by 1,000	use *scp* to copy a single 138MB file
httpd	use ab (apache benchmark) with "ab -n 50000 -c 500 http://localhost/apache_pb2.gif"	

TABLE 3
Performance Data and Standard Deviation (in Parentheses)
for the Original (orig.), Updated Once (upd.once), and Updated Multiple Times (upd.mult) Applications

Application	connection time (ms)			Transfer rate (MB/s)		
	orig.	upd.once	upd.mult	orig.	upd.once	upd.mult
vsftpd	7.63 (0.02)	7.67 (0.02)	8.0 (0.03)	11.62 (0.03)	11.62 (0.03)	11.63 (0.04)
sshd	146.2 (1.8)	146.6 (1.6)	146.6 (1.7)	11.53 (0.01)	11.54 (0.0)	11.54 (0.0)
httpd	112.9 (1.5)	112.4 (1.7)	112.0 (2.9)	11.57 (0.01)	11.58 (0.01)	11.68 (0.03)

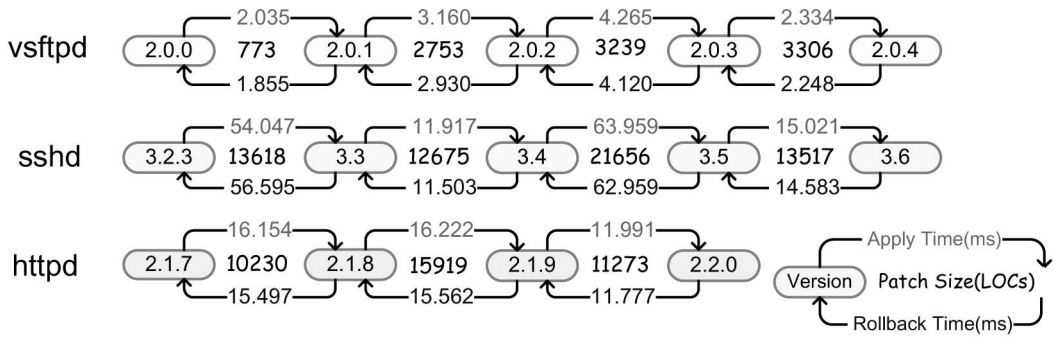


Fig. 8. Update time for the three applications (in milliseconds) and the associated patch size in lines of noncommented code. The applications are under an idle state while being updated.

As shown in Table 3, POLUS incurs undetectable performance overhead, which is much less than the Ginseng [10] system, which uses compiler transformation. For example, according to the performance data, Ginseng incurs 25 percent and 32 percent performance overhead for systems being updated multiple times to the connection time of vsftpd and sshd, accordingly, compared to nearly undetectable overhead in POLUS. This reflects the fact that the only runtime overhead in POLUS is caused by indirect accesses of updated functions. Although the connection time for *vsftpd* increases about 5 percent, we believe such a 0.4 ms difference is, in practice, negligible.

6.2 Update Time

Generally, the update time of an application is decided by the amount of changed types, variables, and affected functions. We measured the update time for applying and rolling back an update. We first updated an application from its first version to its last version and then rolled back to the first version. Fig. 8 shows the update time of our measured three applications and the associated patch size in lines of noncomment code. The update time includes the whole process to apply a patch. It includes loading the patch into memory, doing necessary preparation work and applying the patch, during which there may be some performance degradations to the running systems.

As shown in Fig. 8, the time to apply/roll back an update is modest. All updates are finished within less than 70 ms. Though not exactly, the update time is also related to the patch size, which roughly reflects the number of changes to types, variables, and functions.

6.3 Service Disruptions

To measure the impact of dynamic updates on running services, we used *ab* to issue 15,000 requests for a single 2.4 KB file and collected the throughput of Apache httpd server when an update from version 2.1.7 to 2.1.8 is in progress. Fig. 9 depicts the curve of throughput during the process of updating. There is only a modest amount degradation (about 30 percent) during the update. Further, the time to evolve Apache httpd is still very short, even under a heavy load. Besides, even under a heavy system load, the increase in update time is still modest (from 16.1 ms in Fig. 8 to 22 ms in

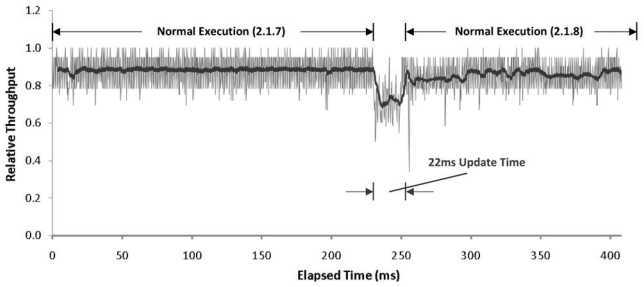


Fig. 9. Impact of live update on *httpd* under a heavy load.

Fig. 9). Therefore, we can conclude that POLUS has very little disruption on the running Apache httpd.

6.4 Memory Overhead

To evaluate the memory overhead incurred by POLUS, we report the memory footprint (both virtual and physical) of the three server applications. For each application, we initialize the POLUS library, insert and apply all of the patches, and roll each application back to its first version. After each operation, we report the memory (both virtual and physical) used by the target application.

As shown in Fig. 10, we find that, for each application, the memory usage (both virtual and physical) increases with the evolution of the target application. The increased memory is composed of two parts—the memory to store functions and data in the patch, and the dynamically allocated memory for heap related data. When there is few dynamically allocated memory (such as vsftpd and sshd), the increase in memory is roughly proportional to the size of the patch. However, for httpd, which requires dynamically allocating data for updated data, the increase in memory is more related to the amount of dynamic memory allocation. We also find that the memory usage does not decrease when rolling back. This is because we do not reclaim the memory used by new functions and data when rolling back.

7 Related Work

A considerable number of systems have been proposed in the literature of dynamic software updating. In this section, we compare our approach with some of those systems.

7.1 Permissive Updating Systems

To support flexible changes in realistic software evolution, dynamic updating systems should be permissive enough to support changes to types, functions (including prototypes) and global variables. Hicks et al. [6] propose a flexible updating system for a type-safe C-like language named Popcorn. Ginseng [10], [19] is a recent system that makes substantial improvement over Hicks et al. [6] to support realistic software updates. Ginseng uses compiler transformation that adds type wrappers and function indirections to make software dynamically updatable. The recent version of Ginseng [19] uses *induced update points*, which requires programmers to specify some safe points and then the compiler analysis will extend the specified safe points to more update points, to support multithreaded applications. In contrast to POLUS, using compiler transformation fails to meet our first criterion of binary compatibility, limiting its application to already running software. Further, the compiler transformation will also add performance overhead due to the per-access indirection to typed variables and functions. However, the associated compiler analysis and transformation make the safety of updates easier to ensure and reason about.

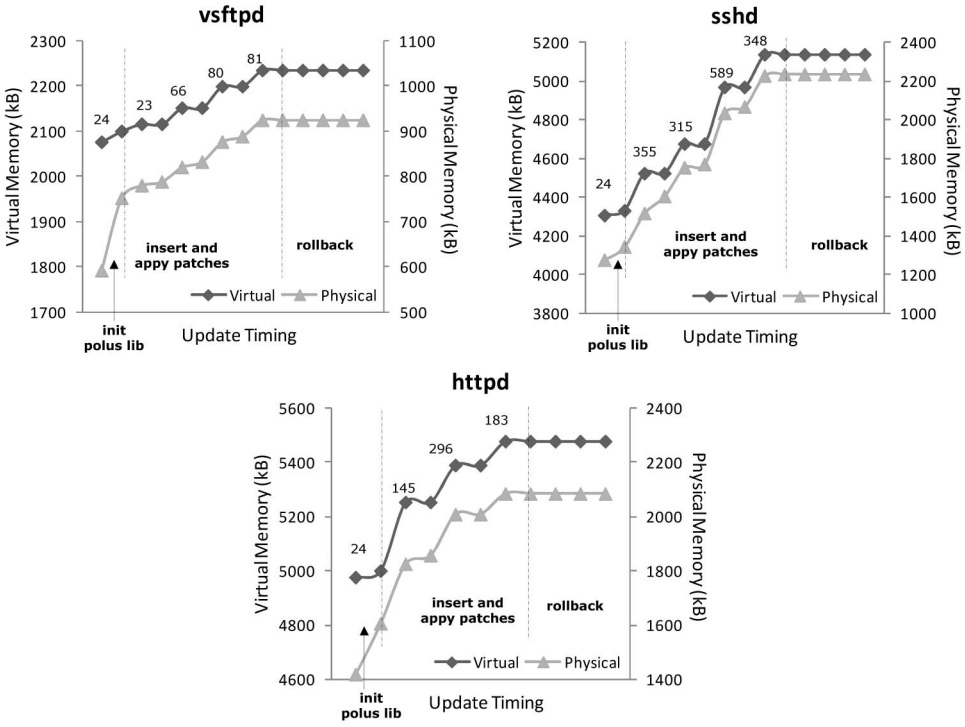


Fig. 10. Memory footprint of measured server applications. Each dot is the memory footprint before applying or rolling back an update. The number above the dot is the corresponding size of the patch in Kbytes (KB).

7.2 Restrictive Updating Systems

Some existing updating systems only support restrictive online updates, which support only code changes [9], [20], cannot support changes to type definitions [5], [21], [22], or is very restrictive in update criteria [23]. For example, OPUS [9] only supports updates not involving global state, while Dyninst [20] only supports runtime code patching, without involving data. Thus, their functionalities are restricted compared to POLUS and are usually used for some stateless security patches and program analysis.

PODUS [5], [21], [22] is an updating system for procedure-based languages (e.g., C and Pascal). In PODUS, updates can only be applied when the procedures to be updated are both syntactically and semantically inactive. Otherwise, the system will retry until the criteria are met. PODUS addresses type-changed functions using *interprocedures*, which is similar to *stub functions* in [6]. In contrast to POLUS, PODUS does not support changes to type definitions and updates to multithreaded applications. Besides, the use of segmented virtual memory for coexisting multiple versions of code also requires special operating systems support and brings potential performance overhead.

DYMOS [23] supports updating programs written in StarMod [24]. In contrast to POLUS, DYMOS requires programmers to manually specify the timing constraints on whether it is safe to issue an update. For example, "*update P, Q, R when P idle*" is a command issued by users in DYMOS requiring that the update of procedure P, Q, and R can only be carried out when procedure P is not on the stack. However, it is not necessary for our system because POLUS is able to issue an update even if P is still active.

7.3 Functional and Object-Oriented Languages

There are also dynamic updating support for systems implemented using functional and object-oriented languages [20], [25], [26], [27], [28], such as standard ML language [25], C++ [26], [29], and Java [27], [30]. The dynamic ML proposed by Gilmore et al. [25] supports replacement of modules at runtime. Dynamic C++ classes [26] use proxy classes to allow the update of code in a running system, and Stanek et al. [29] extends the updatability of C++ classes by supporting fission and fusion of classes and using state transformation to transfer states. Dynamic updating mechanism called dynamic Java classes is also implemented in Java [27] to provide a JVM with updating ability. Besides, Orso et al. [28] used bytecode transformation to make Java programs updatable without the need of a customized JVM in [27].

However, the above systems still lack a certain level of flexibilities. For example, Dynamic ML only permits signature extensions; in other words, updated modules cannot change or remove the elements in their signatures (signatures in ML are similar to type definitions in imperative languages). Furthermore, Dynamic ML performs its code replacement during garbage collection, which means that all the functions to be updated cannot be on the call stack (inactive) when updating. For Dynamic C++ classes [26] and Dynamic Java classes [27], interface changes are disallowed in their system and programmers are supposed to have dynamic updating in mind while they are programming, which breaks the criterion of binary compatibility. Jvolve [30] is a recent system that uses the update-point approach to support flexible updates in Java software evolution, including adding and changing classes. To perform an update, it forces all threads to reach a safe point where no changed methods are being used.

Nevertheless, it is easier to do dynamic update for object-oriented languages than for procedure languages, such as C. For example, some object-oriented languages (e.g., Java and C++) have a function indirection table inherently, which makes them more convenient to update a function. We believe the approach taken by POLUS can be similarly implemented in such high-level languages, resulting in better clarity and less implementation efforts.

7.4 Component-Based Updating Systems

Component-based dynamic software updating has gained research interests because each module is naturally an update unit. Fabry [31] developed a technical framework for dynamic updating in a system made up of modules. He introduces version number to each instance of the data structure and uses a locking mechanism to ensure that the code always executes the correct version of the data. One problem of his framework is that instances cannot be updated until all of the execution of old versions of modules have completed, while, in POLUS, it is not necessary because our synchronization mechanism ensures the consistency when the old and the new versions coexist. Additionally, his framework only supports changes to abstract data types.

Boyapati et al. [32] described a system that supports lazy updating for persistent objects. They introduce upgrade modularity conditions forcing an update ordering so that type safety can be preserved during and after an update (e.g., new objects are not likely to be accessed by old code). They rely on object encapsulation to meet the conditions they present, ensuring a well timing upgrade. They also use transactions to ensure a proper timing for an upgrade. The drawback of their system is that it requires programmers to write highly encapsulated programs so that upgrades can be applied effectively and efficiently. Further, the programs to be updated are supposed to have a database-style transaction feature, which limits its generality.

Gregersen et al. [33] proposed using unanticipated dynamic software updating to overcome the constraint of the reload feature in the NetBeans platform and to support state-reserved dynamic switches among multiple versions of modules. The in-place proxification mechanism is similar to the function indirection used in POLUS, which does not require a wrapper. They also introduced a correspondence handling technique that uses table mapping to cross the version barrier, which is similar to our optimization for multiple function indirection. However, they currently do not support multithreading and updates to fields.

Kramer and Magee [34] introduced the notion of quiescence to ensure system consistency in updating distributed system. However, the quiescence requirement is too strong and might bring nontrivial disruption to systems to be updated. Hence, Vanderwoude and Ebraert [35] proposed a weaker condition, named tranquility, which only requires adjacent nodes with the node to be updated to be in a passive mode, instead of all dependent nodes as that

in quiescence. Oreizy et al. [36] proposed ArchStudio, an architecture-based approach that uses explicit structural model to manage runtime changes. ArchStudio relies on software connectors, which bind components together and mediates intercomponent communications, to implement on-the-fly rebinding of components.

7.5 Formal Methods and Update Safety

There are some theoretical efforts in literature trying to formalize the software update [14], [37], [38] and analyze the safety of updates [15], [16]. Some worked on the validity of an update [37], [38] and others focused on the type safety [14], [15]. However, to the best of our knowledge, most of them are only applicable to single-threaded or update-point-based systems.

Stoyle et al. described Proteus [15], a formal model for dynamic updating C-like languages. It examines the safety of dynamic updates and proposes the notion of *representation consistency*, which avoids runtime type errors. They used this formal model to automatically infer safe update points in the application and to improve their previous work [6]. But their analysis is designed for single-threaded, update-point-based systems. They are extending the standard effect systems with a concept called *contextual effects*, and presenting a novel correctness property called *transactional version consistency* (TVC) [16]. It addresses version consistency problems in dynamic updating network servers (e.g., vsftpd). However, their analysis is still for single-threaded, update-point-based software.

Gupta et al. [37], [38] presented a formal framework to model the online software changes for C-like languages. The authors prove that determining the validity of an online change is generally undecidable. They also developed several conditions that are sufficient to achieve a valid online change. A simple program model without procedures is considered first. Later, the authors extended it to a procedure-based program model. In their working prototype, dynamic updating is performed by putting the new code in a new process and transferring states from the old process to the new process. However, their system is unable to update active code, and the framework is based on single threaded programs.

Duggan [14] presented a type-based approach to hot swapping running modules using formal methods. It permits the representation of named types to be changed on the fly by allowing multiple versions of types to coexist in the system. He introduced box types which are branded with runtime tags using *fold* operations and allow them to be casted into different versions of types by using *unfold* operations. If necessary, *version adapters* are used to perform type coercion between different versions of types. This approach ensures well-formed updates and is quite similar to that in [15].

7.6 Updating Operating Systems

K42 is an object-oriented operating system built with dynamic update supports [39], [40], [41]. Implementing dynamic update in K42 is much easier as K42 is built with many good features, such as building blocks, thread generation counts, and object translation tables, their system supports interposition and hot-swapping components in the operating system. In contrast to POLUS, K42 uses an update-point-based approach, which requires a component should be in quiescence before it can be hot swapped, which is sometimes hard to achieve in a threaded environment, especially in an operating system.

Ksplice [11] is a restrictive updating system similar to OPUS [9], but targets the Linux kernel. Like OPUS, Ksplice can also only support updates that do not involve changes to global state, which restricts its flexibility but makes it easier to ensure patch safety.

LUCOS [42] might be the most similar system to POLUS. However, LUCOS supports on-the-fly updates to contemporary operating systems using system virtualization techniques. In our current work, we apply similar concepts to application software, yet without an additional virtualization layer as in LUCOS. Further, to reduce the tedious work in patch construction, we provide compiler support to automate most of the work.

8 Conclusion and Future Work

Providing dynamic updating to application software is hard, especially for realistic software updates that involve flexible changes. This situation becomes even worse when facing multithreaded software. This paper analyzed that the major difficulty in dynamic updating such software is due to the ubiquitous yet complicated thread interactions, which makes it extremely difficult to find a quiescent state to apply an update, and thus increases the possibility to unduly delay a critical update.

In this paper, we have overcome the difficulty in dynamic updating multithreaded software by using a relaxed consistency model. It allowed online updates to active code and data, and maintained the state consistency using a "bidirectional write-through" state synchronization protocol. We showed that relaxing the quiescence requirement in dynamic updating naturally suited the prevalent multithreading programming model, in which the code and data to be updated are usually be referenced by multiple threads and can hardly be quiescent. To ensure system consistency, our approach additionally required the states to be bidirectional convertible between the old and new version of code.

Besides the natural support for multithreaded software, the advantages of dynamic updating using a relaxed consistency model are mainly two-fold. First, there is no timing issue such as that in previous approaches, as an update can be applied even if the code and data to be updated are active. Second, it does not require changing the layout of code and data structures, thus retaining backward compatibility.

We have shown that the relaxed consistency model can be implemented for flexible languages, such as C, on top of commodity hardware and software stack. Our prototype system, POLUS, was designed to support flexible changes in real-world updates—changes to types, global variables, and functions. With POLUS, permissive updates to realistic multithreaded software can be applied on the fly with very little performance overhead, yet without the updating timing restriction and retains the backward compatibility. Our experiences in dynamic updating three prevalent

software systems over a relative long period have confirmed that POLUS is effective and flexible enough in applying dynamic software updates to real, large, and complex software systems with minimal service disruption.

While we have shown our approach is practical for C-like languages, in our future work we intend to apply the update model and protocol to object-oriented systems such as JVM, which are presumably much easier to implement our approach due to the good modularity. Moreover, current implementation of POLUS retains binary compatibility that allows it to support legacy systems or currently running software. Doing so complicates the implementation of POLUS and makes it difficult to handle some infinite loops during update (although updates to the calling functions of such loops are rather rare in practice). For newly developed software, we plan to use compiler transformations (as in [10]) such as look extraction and function call wrapping to make the programs more friendly to POLUS, thus making the implementation of POLUS simpler and reducing some update-time overhead. Finally, we also plan to investigate ways to provide a formal proof on the safety of the relaxed consistency model and the "bidirectional write-through" synchronization protocol in POLUS.

ACKNOWLEDGEMENTS

This work was funded in parts by China National 973 Plan under grant numbered 2005CB321905, Shanghai Leading Academic Discipline Project (Project Number: B114), and a research grant from Intel numbered MOE-INTEL-09-04. The source code of POLUS and evolutionary tests are available with sourceforge (http:// sourceforge.net/projects/polus/). The patch generator is distributed under the BSD License, others are based on GNU General Public License. A preliminary and abridged version of this paper was presented at the 29th International Conference on Software Engineering (ICSE '07).

REFERENCES

[1] I. Neamtiu, J. Foster, and M. Hicks, "Understanding Source Code Evolution Using Abstract Syntax Tree Matching," *ACM SIGSOFT Software Eng. Notes,* vol. 30, no. 4, pp. 1-5, 2005.

[2] Microsoft Corp. "Windows Update and Automatic Updates," http://windowsupdate.microsoft.com/, 2007.

[3] D.E. Lowell, Y. Saito, and E.J. Samberg, "Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance," *ACM SIGOPS Operating Systems Rev.,* vol. 38, no. 5, pp. 211-223, 2004.

[4] D.A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Technical Report UCB//CSD-02-1175, Univ. of California, Mar. 2002.

[5] O. Frieder and M.E. Segal, "On Dynamically Updating a Computer Program: From Concept to Prototype," *J. System Software,* vol. 14, no. 2, pp. 111-128, 1991.

[6] M. Hicks and S. Nettles, "Dynamic Software Updating," *ACM Trans. Programming Languages and Systems,* vol. 27, no. 6, pp. 1049-1096, 2005.

[7] M. Barber, "Increased Server Availability and Flexibility through Failover Capability," *Proc. 11th USENIX Conf. System Administration,* pp. 89-98, 1997.

[8] D. Pescovitz, "Monsters in a Box," *Wired,* vol. 8, no. 12, pp. 341-347, 2000.

[9] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online Patches and Updates for Security," *Proc. USENIX Security,* pp. 287-302, 2005.

[10] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical Dynamic Software Updating for C," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 72-83, June 2006.

[11] J. Arnold and F. Kaashoek, "Ksplice: Automatic Rebootless Kernel Updates," *Proc. EuroSys,* 2009.

[12] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A Powerful Live Updating System," *Proc. Int'l Conf. Software Eng.,* pp. 271-281, 2007.

[13] B. Demsky and M. Rinard, "Goal-Directed Reasoning for Specification-Based Data Structure Repair," *IEEE Trans. Software Eng.,* vol. 32, no. 12, pp. 931-951, Dec. 2006.

[14] D. Duggan, "Type-Based Hot Swapping of Running Modules," *Proc. Sixth ACM SIGPLAN Int'l Conf. Functional Programming,* pp. 62-73, 2001.

[15] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "*Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating," *ACM Trans. Programming Languages and Systems,* vol. 29, no. 4, 2007.

[16] I. Neamtiu, M. Hicks, J.S. Foster, and P. Pratikakis, "Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming," *Proc. ACM Conf. Principles of Programming Languages,* pp. 37-49, 2008.

[17] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, "Cil: Intermediate Language and Tools for Analysis and Transformation of C Programs," *Proc. Int'l Conf. Compiler Construction,* pp. 213-228, 2002.

[18] M.A. Linton, "The Evolution of Dbx," *Proc. Usenix Summer,* pp. 211-220, 1990.

[19] I. Neamtiu and M. Hicks, "Safe and Timely Dynamic Updates for Multi-Threaded Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* June 2009.

[20] B. Buck and J.K. Hollingsworth, "An API for Runtime Code Patching," *J. High Performance Computing Application,* vol. 14, no. 4, pp. 317-329, 2000.

[21] M.E. Segal and O. Frieder, "Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime," *J. Software Maintenance,* vol. 1, no. 1, pp. 59-79, 1989.

[22] M. Segal and O. Frieder, "On-the-Fly Program Modification: Systems for Dynamic Updating," *IEEE Software,* vol. 10, no. 2, pp. 53-65, Mar. 1993.

[23] I. Lee, "Dymos: A Dynamic Modification System," PhD dissertation, The Univ. of Wisconsin, 1983.

[24] R.P. Cook, "*MOD a Language for Distributed Programming," *IEEE Trans. Software Eng.,* vol. 6, no. 6, pp. 563-571, Nov. 1980.

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

[26] G. Hjálmtýsson and R. Gray, "Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program," *Proc. USENIX Ann. Technical Conf.,* pp. 65-76, 1998.

[27] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes, "Runtime Support for Type-Safe Dynamic Java Classes," *Proc. 14th European Conf. Object-Oriented Programming,* pp. 337-361, 2000.

[28] A. Orso, A. Rao, and M. Harrold, "A Technique for Dynamic Updating of Java Software," *Proc. IEEE Int'l Conf. Software Maintenance,* 2002.

[29] J. Stanek, S. Kothari, T. Nguyen, and C. Cruz-Neira, "Online Software Maintenance for Mission-Critical Systems," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 93-103, 2006.

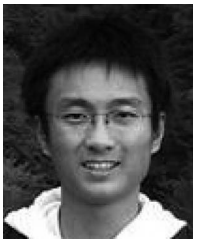
[30] S. Subramanian, M. Hicks, and K.S. McKinley, "Dynamic Software Updates: A VM-Centric Approach," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2009.

[31] R.S. Fabry, "How to Design a System in which Modules Can Be Changed on the Fly," *Proc. Second Int'l Conf. Software Eng.,* pp. 470-476, 1976.

[32] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman, "Lazy Modular Upgrades in Persistent Object Stores," *Proc. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 403-417, Oct. 2003.

[33] A. Gregersen and B. Jorgensen, "Module Reload through Dynamic Update: The Case of NetBeans," *Proc. 12th European Conf. Software Maintenance and Reeng.*, pp. 23-32, 2008.
[34] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Software Eng.*, vol. 16, no. 11, pp. 1293-1306, Nov. 1990.
[35] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquillity: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Trans. Software Eng.*, vol. 33, no. 12, pp. 856-868, Dec. 2007.
[36] P. Oreizy, N. Medvidovic, and R. Taylor, "Architecture-Based Runtime Software Evolution," *Proc. Int'l Conf. Software Eng.*, pp. 177-187, 1998.
[37] D. Gupta, P. Jalote, and G. Barua, "A Formal Framework for On-Line Software Version Change," *IEEE Trans. Software Eng.*, vol. 22, no. 2, pp. 120-131, Feb. 1996.
[38] D. Gupta, "On-Line Software Version Change," PhD dissertation, Indian Inst. of Technology Kanpur, Nov. 1994.
[39] C.A.N. Soules, J. Appavoo, K. Hui, R.W. Wisniewski, D.D. Silva, G.R. Ganger, O. Krieger, M. Stumm, M.A. Auslander, M. Ostrowski, B.S. Rosenburg, and J. Xenidis, "System Support for Online Reconfiguration," *Proc. USENIX Ann. Technical Conf.*, pp. 141-154, 2003.
[40] A. Baumann, G. Heiser, J. Appavoo, D.D. Silva, O. Krieger, R.W. Wisniewski, and J. Kerr, "Providing Dynamic Update in an Operating System," *Proc. USENIX Ann. Technical Conf.*, pp. 279-291, Apr. 2005.
[41] A. Baumann, J. Appavoo, R.W. Wisniewski, D. Da Silva, O. Krieger, and G. Heiser, "Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly," *Proc. USENIX Ann. Technical Conf.*, 2007.
[42] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live Updating Operating Systems Using Virtualization," *Proc. Second Int'l Conf. Virtual Execution Environments*, pp. 35-44, 2006.

Haibo Chen received the BSc and PhD degrees in computer science from Fudan University in 2004 and 2009, respectively. He is currently an assistant professor in the Parallel Processing Institute, Fudan University, where he coleads the system software group. His research interests include software evolution, system software, and computer architecture. He is a member of the IEEE and the IEEE Computer Society.



Jie Yu received the BSc degree in software engineering from Fudan University in 2007. He is currently a PhD student in the Computer Science and Engineering Department at the University of Michigan Ann Arbor. His research interests include software evolution, programmability and reliability of parallel systems, and concurrent software testing. He is a student member of the IEEE.

Chengqun Hang received the BSc degree in software engineering from Fudan University in 2009. He is now a software engineer with Microsoft.

Binyu Zang received the PhD degree in computer science from Fudan University in 1999. He is currently a professor and the director of the Parallel Processing Institute, Fudan University. His research interests include compilers, computer architecture, and systems software.

Pen-Chung Yew received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1981. He is currently a professor in the Department of Computer Science and Engineering, University of Minnesota. His major research interests include multicore architectures, compilation techniques, and OS for multicore embedded systems. He is a fellow of the IEEE and a member of the IEEE Computer Society.