State Transfer for Clear and Efficient Runtime Updates

Christopher M. Hayden, Edward K. Smith, Michael Hicks, Jeffrey S. Foster
University of Maryland, College Park
{hayden, tedks, mwh, jfoster}@cs.umd.edu

***Abstract*—Dynamic software updating (DSU), the practice of updating software while it executes, is a lively area of research. The DSU approach most prominent in both commercial and research systems is *in-place updating*, in which patches containing program modifications are loaded into a running process. However, in-place updating suffers from several problems: it requires complex tool support, it may adversely affect the performance of normal execution, it requires challenging reasoning to understand the behavior of an updated program, and it requires extra effort to modify program state to be compatible with an update.**

This paper presents preliminary work investigating the potential for *state transfer updating* to address these problems. State transfer updates work by launching a new process running the updated program version and transferring program state from the running process to the updated version. In this paper, we describe the use and implementation of Ekiden, a new state transfer updating library for C/C++ programs. Ekiden seeks to redress the difficulties of in-place updating, and we report on our experience updating VSFTPD using Ekiden. This initial experience suggests that state transfer provides the availability benefits of in-place DSU approaches while addressing many of their shortcomings.

I. Introduction

In recent years, dynamic software updating (DSU) systems, which enable software updates to take place at runtime, have evoked a flurry of interest and activity. Research DSU systems for C, C++, and Java have been used to dynamically update servers and operating systems [4], [15], [8], [12], [22]. Ksplice [6] and Unsanity's Application Enhancer [23], each commercial offerings of binary-level DSU, provide runtime updating support to the Linux Kernel and to Mac OS X applications, respectively. Ericsson's Erlang programming language [5] has DSU support as a core feature, and its creators laud the uptime that DSU has enabled for fielded telecommunications systems.

Each of these systems employs *in-place updating*: when a new program release is available, its developers write a *dynamic patch* that contains the new code. This patch is loaded into the running program, and execution of the old code is redirected to the new. Such redirection is either enabled by special compilation of the original program, as in Ginseng [15], Erlang, and UpStare [13], or forced on the running system via runtime program rewriting, as in Ksplice [6], POLUS [8], or Jvolve [22]. Patches also include *state transformation* code that runs after the patch is loaded, adjusting both data and control state.

While popular, in-place updating has several drawbacks: (1) it requires a compiler or binary rewriter that can be complex and tricky to get right, and may not work in some contexts, such as when a memory region containing executable code is read-only; (2) compilers to support in-placing updating sometimes use static analyses to ensure that program transformations are semantics-preserving, but such analyses are inherently conservative and so will effectively force some programs to be altered before they can be correctly compiled; (3) the mechanisms used to implement in-place updating often introduce steady-state run-time overhead; e.g., inserted levels of indirection add extra instructions and inhibit compiler optimizations like inlining; (4) reasoning about the behavior of an updated program places additional cognitive burden on the programmer, particularly when updates could take effect from many run-time states; and (5) all state of the new program must be explicitly initialized, even state that does not carry over from the old version. In our experience, point (4) is the real show-stopper: It takes a heroic effort to simultaneously reason about a program, the points within that program where an update might occur, the modified code in the patch, and the state transformation code.

In this paper we argue that these drawbacks can be addressed by using *state transfer-based updating* [10]. We have been developing a library, called Ekiden, that implements this approach for C and C++ programs.[1] In Ekiden, updates work by starting the new program when an update is requested, marshaling and transferring state from the old version of the running program to the new, transforming the state in the new program, and then terminating the old program.

Ekiden's approach addresses many of the disadvantages of in-place updates: (1) no complicated program transformations are needed—state transfer updating can be implemented as a library that builds largely on existing infrastructure; (2) the approach does not rely on static analyses, so no programs are conservatively rejected; (3) using state transfer does not inhibit compiler optimizations and imposes no steady-state overhead—the only slowdown occurs at update time, a relatively rare event; (4) while the effort of preparing an updatable program is slightly increased, the cognitive burden of reasoning that an update is correct is dramatically decreased since Ekiden forces us to indicate exactly where and how it will take place; and (5) the effort in writing state transformers is reduced, since new state is automatically initialized by the

[1] Ekiden is named after the Japanese long-distance relay race, suggesting both the desired long running time of dynamically updated programs and the hand-off of state that takes place between versions.

new version of the program, so only old state that evolves during execution needs to be transferred and updated.

On the other hand, state transfer is not a panacea, and presents several challenges: (1) while steady-state overhead is minimized, the cost to transfer large volumes of state could be high; (2) some types of state (e.g., state internal to a library) may be difficult to transfer; (3) not all applications can have multiple instances running at once, e.g., operating systems (typically); and (4) more manual effort is required to determine which state to transfer, to insert annotations into the program (cf. Section II), and possibly to write serializers for transferred state (though we suspect this can all be automated in most cases). While we have yet to thoroughly evaluate these trade-offs, we believe that for many applications, the benefits outweigh the drawbacks.

In this paper, we present the design of Ekiden, report on our preliminary experience using Ekiden to update VSFTPD, a server we have previously experimented with using Ginseng, and compare Ekiden to in-place updating approaches generally. In our experience so far, it has been straightforward to implement state transfer for VSFTPD using Ekiden, particularly because Ekiden did not require us to run and satisfy a complex program analysis like Ginseng's.

II. Runtime Upgrades Using State Transfer

In this section, we give an overview of how Ekiden works, what program changes it requires from users, and our experience using Ekiden to update VSFTPD.

A. Preparing a Program to use Ekiden

To make use of Ekiden's capabilities, the programmer needs to make some changes to their source code, as illustrated in Figure 1. The left of the figure shows a minimal, single-threaded server program, and the right shows the required modifications (some have been slightly simplified for improved illustration).

Tagged Program State: The programmer must identify the critical state to be transferred at update time. Some state need not be transferred if it will be correctly initialized during new-version start-up or can be derived from other state. In Figure 1, the variable nclients, which tracks the number of clients that have connected, has been tagged using the function tag_st() so that it will be transferred. The variable const_val is not tagged since it is never written once initialized; as such we can rely on the new program to simply initialize it when the program restarts.

Update Points: An update is triggered when a new program version becomes available and is noticed during a call to the Ekiden function update_point; programmers must insert calls to this function in their program. The update_point function calls fork to create a new child process, and then calls exec to start the new program version. This new version arranges to transfer the tagged state from the old program version via a UNIX domain socket. When all state is transferred, the old process shuts down.

```
void handle_client(int cnt,
 int val) {
  while (1) {

    // process client
    // requests
  }
}
int main(int argc,
 char **argv) {
  int nclients = 0;
  int const_val = 42;

  ...

  while (1) {

    // accept connections
    handle_client(nclients,
      const_val);
    nclients++;
  }
}
```

(a) Original server

```
void handle_client(int cnt,
 int val) {
  while (1) {
   update_point("cl_loop");
    // process client
    // requests
  }
}
int main(int argc,
 char **argv) {
  int nclients;
  int const_val = 42;
  tag_args(argc, argv);
  if (tag_st(&nclients,
             import_int,
             export_int)) {
   // not updating
   nclients = 0;
  }
  ...
  if (upd_from("cl_loop")) {
   // enter client loop
   handle_client(client_count,
    const_val);
  }
  while (1) {
   update_point("con_loop")
   // accept connections
   handle_client(nclients,
    const_val);
   nclients++;
  }
}
```

(b) Modified server

Fig. 1. Preparing a program for state transfer updating

We advocate placing a single update point at the beginning of each long-running event-processing loop in the program, as seen in the two update points labeled "cl_loop" and "con_loop" in Figure 1. At such update points, the server is between events and hence tends to be "quiescent," which typically makes it easy to map program state between the old and new versions. Of course, in general update points may be placed anywhere, but incautious placement of update points can make updates harder to reason about.

Updating Control Flow: When the new version of a program starts up during an update, it must ultimately resume execution from where the old version left off. This may require modifying the control flow of the program to branch to update points after the new version is launched. In Figure 1, for example, we modified the start of main to jump directly to the handle_client loop if the incoming state indicates the update occurred at the update point labeled with "cl_loop." Assuming there are only a few update points, we suspect that such modifications will be straightforward for most programs, but more experience is needed to be sure.

Serialization: Any tagged state must be transferred between the old and new program. Tagged state must be properly serialized when sending and deserialized on receipt. To specify

```
struct foo {
  char* ZT name;
  char* PTRARRAY(buf_len) buf;
  int buf_len;
  struct foo* IGNORE misc;
  struct foo* PTR next;
};
```

Fig. 2. Data structure annotated for serialization generation

how serialization should take place, the tagging function tag_st takes as arguments pointers to the serializer and deserializer functions (along with a pointer to the state to transfer). If a program is started as an update from a previous version, the tag_st function initializes its first argument with a value that is deserialized, and tag_st returns false. Otherwise, tag_st returns true. Looking at Figure 1, we can see that the value of nclients is serialized with export_int when an update takes place, and will be deserialized using import_int. If the new program is started from scratch, the tag_st call will return true, so nclients is set to 0.

The functions export_int and import_int are part of an Ekiden library for serializing basic C types, and this library can be also used as a basis for serialization code for user-defined types. (State transfer implementations for other languages, including Java and Ruby, could leverage language support for marshaling).

To reduce the burden of writing serializers by hand, we have developed a tool that generates serialization code based on programmer annotations, where annotations indicate properties like the sizes of arrays and fields that should not be transferred. Figure 2 shows a simple example of a data structure, foo, that we have annotated for use with our generator. The name field has been annotated using ZT as a pointer to a null-terminated string; buf has been annotated using PTRARRAY(buf_len) as a pointer to an array containing buf_len chars; buf_len requires no annotation as it is a simple integer; misc has been annotated with IGNORE, indicating that it should not be serialized; and next has been annotated as a pointer (PTR) to a single foo instance (the PTR annotation is the default treatment for pointers, so this annotation is optional). These annotations were influenced by the ones used by the Deputy compiler [9] to prevent memory errors. Our serialization tool generates code that serializes each object in memory once, to allow for data structures containing cycles. We have found, in early experiments, that this tool makes the creation serialization code quite easy. However, we will continue to refine this tool and consider alternatives based on our experience updating additional programs.

We would also like to transfer open file descriptors during updates, e.g., so the new version can continue listening on open sockets and communicating with connected clients. Fortunately, because exec does not close open file descriptors, programs using Ekiden can simply pass along the file descriptors' integer values (using export_int), since they will have the same meaning in the parent and child processes. Alternatively, file descriptors could be transferred using UNIX domain sockets and the sendmsg() system call [21]. Libraries such as the Ancillary Library [1] and Facebook's libafdt [2] simplify this process.

We may also wish to transfer function pointers, but how to do so is not necessarily obvious: we cannot simply pass the pointers as is because functions may not have the same address in both versions (indeed, some functions may not even persist across versions). At the moment, the best alternative is to map function addresses to integers or strings during serialization and map them back during dserialization. We hope to streamline this process in the future by adding function pointer annotations to our serialization generation tool.

B. Experience Updating vsftpd

We have used Ekiden to construct a sequence of four updatable versions of VSFTPD [24] (1.1.3, 1.2.0, 1.2.1, and 1.2.2), representing two years of changes. Adding updating support to each version of VSFTPD required only 16 discrete changes to 3 out of 30 source files. Here, we discuss the nature of these modifications and what we learned from these early experiments.

We made minor changes to VSFTPD to configure the updating framework. First, we needed to install a UNIX signal handler to initiate an update upon receipt of a particular signal. Second, we added code to save the command-line arguments used to invoke the current version so they can be applied when starting the new version. Finally, we indicated the file system path where update_point should look when signaled that a new program executable is available.

We added two update points to VSFTPD, each to be reached between iterations of long-running loops, basically following the example in Figure 1. The first update point was added to the loop that accepts connections from new clients. The default start-up control flow of the program enters this loop following some initialization, so no modifications were required to initiate entry to this loop following an update. A second update point was added to the loop that processes FTP commands from connected clients. In this case, we inserted some code to jump to this command-processing loop after an update from the corresponding loop in the old version. This change required only a slight modification of control flow, to avoid entering the client acceptance loop and to enter the FTP command loop. It was also necessary to wrap an existing block of code in a condition to prevent re-display of an FTP banner that should display once per connection.

VSFTPD required serialization and transfer of only two **struct** types, and only five state items were tagged in total. The VSFTPD process that accepts new client connections maintains three essential pieces of state: a hash table mapping process IDs to connected IP addresses, a hash table mapping IP addresses to connection counts, and a count of children. VSFTPD processes that handle connected clients each maintain a single record containing all critical state for the client. In total, updating VSFTPD with Ekiden required identifying four program variables for transfer. The only non-obvious task was

determining how to write the serializer/deserializer functions for the hash tables, since their representation contains function pointers (the hash functions). We simply hardcoded the choice of these functions to the ones used in our application, using an enumeration. This solution is somewhat unsatisfying because the serializers for a generic type needed to be aware of each of the type's uses. We are working on refinements to our serialization generation tool to improve handling of this case.

We found that, overall, most state in VSFTPD need not be transferred, as it is initialized during server start-up and only read during subsequent execution. While identifying and annotating state for transfer was easy, the code written to serialize and transform it accounted for the bulk of the modification effort. Further refinements to our serialization-generation tool should reduce this burden. The developer must still write the code to transform old program state to conform to the new version, but this effort is manageable since most state was not changed between versions. Future work will determine whether the same is true for other programs.

During the process of configuring VSFTPD for updating, we found it useful to construct and test "updates" from one version to the same version. This testing allowed us to check that the state necessary to resume execution was transferred.

In earlier work, we prepared VSFTPD for updating using Ginseng. Interestingly, that work required a different set of modifications. As with Ekiden, we explicitly annotated update points. To ensure type-safety, Ginseng's analysis also required annotating certain types as non-updatable. To enable updates in long-running loops to reach the updated version of the loop-body and the code that follows the loop, we added annotations directing Ginseng to extract particular code blocks into separate functions. These requirements added up to 14 manual changes per version. This count does not include several minor changes that were required to ensure compatibility with Ginseng's analysis. While it is difficult to directly compare the effort required to update VSFTPD using Ginseng versus Ekiden, the most notable difference is that Ekiden did not require us to modify VSFTPD to satisfy a complex program analysis.

III. State Transfer versus In-place Updating

In this section, we describe how state transfer updating addresses the drawbacks of in-place updating that we enumerated in the introduction.

Complex Tool Support: In-place DSU approaches frequently rely on complex compilation schemes and program analyses to support updating. Figure 3 shows the updating process for Ginseng, in which a non-standard compiler instruments the program for updating and prepares patches. Ginseng also uses a complex safety analysis to ensure that updates will not violate type safety. As we described in the previous section, preparing VSFTPD for Ginseng required annotating certain types as non-updatable. State transfer updating is more transparent and imposes far fewer restrictions. In our experience, the non-standard compilation schemes and safety analyses used by in-placing updating systems are undesirable

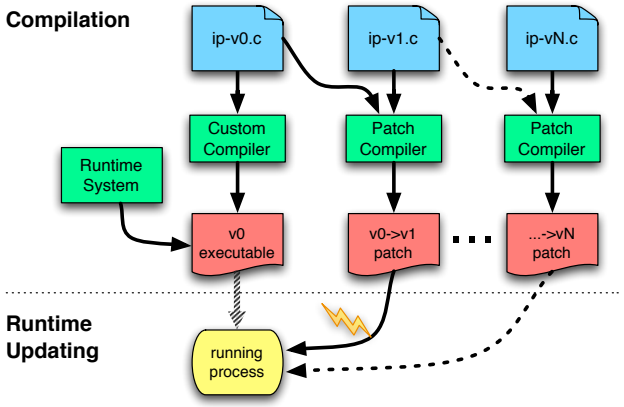

Fig. 3. In-place Update Compilation and Updating

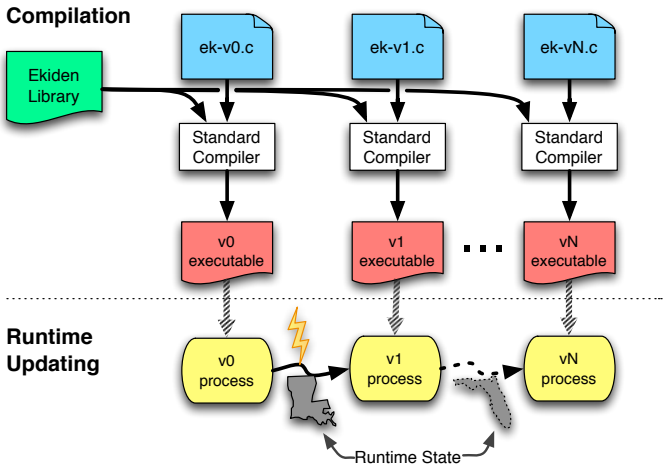

Fig. 4. Ekiden Compilation and Updating

because they introduce new points of failure and behave in ways that are hard to understand.

Because Ekiden is implemented as a library, the update development process, illustrated in Figure 4, need not involve any non-standard tools. While as mentioned above we have developed tools to automate generation of serialization and transformation code, these tools are entirely optional. Moreover, we have found they are easy to understand, since they depend only on type definitions in the program, and they also can handle programs that use odd or unsafe idioms (e.g., those involving typecasts, pointer aliasing, and other features that often confuse static analyses).

Steady-state Performance: In-place updating schemes often introduce a level of indirection so that function calls and variable accesses will reach the appropriate version. For example, Ginseng compiles the program so that for each function `f`, a global variable `f_ptr` is introduced, initialized to `f`. Direct calls to `f` occurring in the program are changed to calls via `f_ptr` instead. When the update takes place, `f_ptr` is redirected to loaded code that implements the new version of `f`. Erlang [5] and K42 [12] are similar. To reduce the steady state overhead, some systems, like POLUS [8] and

Ksplice [6], rewrite the old version of a function within the running program's code to call the new version instead. Doing so requires inserting a "trampoline," which is another kind of indirection.

Adding indirection has two potential downsides: the indirection itself may incur a performance penalty during normal execution, and it may also require disabling compiler optimizations that would remove the indirection. Both may hinder an updatable program's performance. Because Ekiden creates a new process to run the updated code, it makes no assumptions about the internal organization of the program. As a result, no indirection is required, and Ekiden is compatible with all compiler optimizations. For programs where steady-state performance is critical, the Ekiden approach provides a significant advantage. Trampolines have the additional downside that they are incompatible with useful, increasingly common measures to improve security. In particular, use of trampolines requires setting the code portion of memory to be writable, which opens the system up to rootkits [16].

Developer Burden: Reasoning about the behavior of a program updated using the in-place approach may be challenging, since it requires simultaneously considering when updates might happen, the contents of the patch, and whether the resulting execution of code at different versions is correct.

In-place DSU approaches load a patch containing modified functions and typically ensure that future calls to those functions reach the updated version. This means that the body of a particular function is executed entirely at the version at which it was invoked, but may call updated code. This policy may produce undesirable behavior in two ways. First, it prevents execution from reaching the updated version of a function that was active on the stack when the update occurred. In our experience with Ginseng, we observed that this policy often prevents the updated body of a loop (and code following the loop) from being reached when an update occurs within the loop, and solving this problem requires extracting these code blocks (via Ginseng-supported annotations) into new functions. Second, this policy may cause two pieces of related code to be executed at different versions, producing a *version consistency error*. We have previously observed that these errors do occur and are not always prevented by *safety checks* designed to prevent some problematic update timings [11].

Both in-place and state-transfer DSU techniques require modifying a program to support updating. The difference is that the code for state transfer makes updating behavior *explicit*, while the changes to a program to support in-place DSU cannot be understood in terms of normal control flow, i.e., they are typically code restructurings made in anticipation of likely function changes in future updates.

Not all in-place updating approaches are created equal, however. A recently proposed approach called *stack reconstruction* permits the active stack and program counter (PC) of the running program to be updated to resume execution at the proper position in the new code. This model is used by UpStare [13] and DynAMOS [14]. Assuming that the stack and PC are updated correctly, stack reconstruction provides similar behavior to Ekiden's but requires fewer manual modifications, e.g., code to jump to equivalent control positions in the new program. But this flexibility has costs. First, the code to capture and restore the stack is introduced by a special compiler, and thus imposes steady-state overhead. Second, the programmer must match up potentially arbitrary stack layouts of the two program versions; inferring the required stack and PC mappings remains an open research question [7]. That said, if UpStare updates were limited to explicit update points, we believe that the reasoning burden would be comparable to that of Ekiden updates.

State Transformation: When updating programs in place, we must provide state transformation code to update all program state that has changed. We have identified two drawbacks with this requirement. First, bug-fixes for some errors, such as memory or resource leaks, may be difficult to fix through in-place updating, since there may be no reference to the leaked resources. By creating a new process to replace the old one, state transfer provides fresh start. Second, some program state (e.g., a table of server commands) is written during initialization and is static thereafter. While changes to such data require manual state transformation under the in-place update approach, with state transfer-based updating such changes occur with no extra effort during startup of the new version. In Neamtiu et al.'s updates to VSFTPD and OPENSSH, a significant portion of the state transformation code was devoted to updating this kind of write-once state [15].

IV. Related Work

State transfer-like techniques are currently used to update many types of programs. As an example, both Apple's iOS and Google's Android mobile platforms persist the state of applications that become inactive so they can be resumed later. When an application is upgraded, the new version must make sense of any state stored by the prior version and begin execution at an appropriate point. In effect, such applications have performed a state transfer-based upgrade.

Process migration techniques support transferring a running program, along with its state, to a different machine. Migration is often performed to improve performance, reliability, or for load balancing. Smith provides a survey of the mechanisms used for migration [20]. Process migration schemes are relevant to state transfer largely because they share the challenge of quickly transferring program state checkpointed either manually or automatically. However, state transfer updating differs from the general problem of process migration in that it must cope with underlying changes to the code and state of the program.

Checkpointing may be implemented in a variety of ways. The libckpt library supports program checkpointing in user-space [17]. Checkpointing may also be implemented at the operating system [19] or virtual machine [3] level to allow handling of program state managed by the operating system.

Potter et al. [18] perform operating system upgrades by migrating running applications to an updated operating system

instance. They facilitate process migration by running each application within *pods*—isolated environments that provide a virtual machine abstraction. These can be viewed as state transfer updates where the state of the updated operating system is its running applications.

Gupta et al. [10] developed a state transfer-based updating system that provides similar functionality to in-place DSU. Updates at arbitrary program points are allowed, subject to an activeness check performed using ptrace. The contents of the stack, heap, and registers are transferred from the old version to the new version. The program counter may need to be adjusted if functions have changed location in memory. Additions of global variables or fields in structures require that padding was present in the original program; if the program versions use different amounts of padding or become misaligned, memory locations will be misinterpreted.

Gupta et al.'s work represents an interesting point in the spectrum of state transfer approaches. Like an in-place updating system, they seek to automate as much of the process as possible. In doing so, they employ mechanisms and policies that may obscure the runtime behavior of updates from the developer. In our research, in contrast, we aim to make behavior of updates understandable and predictable.

V. Conclusion and Future Work

We believe dynamic software updating using state transfer has the potential to gain traction in the developer community in a way that in-place updating techniques have not. In particular, the potential advantages of state transfer are that the behavior of an updated program is explicit, making reasoning about correctness easier; the technique can be implemented as a library that is compatible with developers' existing toolchains; and steady-state performance should be largely unaffected by modifications to support updates.

So far, our experience applying Ekiden to VSFTPD has been positive, but we need more experience and evaluation, including direct comparison with other updating systems. We also need to develop techniques and tools that address challenging cases that may seem to favor in-place DSU techniques, such as updating programs that maintain a large amount of state, transferring state maintained within library code, and supporting updates to concurrent software. Currently, we are working to implement Ekiden updates for additional server programs to discover additional benefits and challenges of the approach.

References

[1] Ancillary library, May 2010. http://www.normalesup.org/~george/comp/libancillary/.

[2] libafdt, May 2010. http://github.com/facebook/hiphop-php/tree/master/src/third_party/libafdt/.

[3] VMWare, May 2010. http://www.vmware.com.

[4] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *USENIX Security*, 2005.

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

[6] Jeff Arnold and Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Eurosys*, 2009. To appear.

[7] Rida A. Bazzi, Kristis Makris, Peyman Nayeri, and Jun Shen. Dynamic Software Updates: The State Mapping Problem. In *The 2nd ACM Workshop on Hot Topics in Software Upgrades (HotSWUp '09)*, October 2009.

[8] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *ICSE*, pages 271–281, 2007.

[9] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *In European Symposium on Programming*, 2007.

[10] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software Practice and Experience*, 23(9):949–964, September 1993.

[11] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. A testing based empirical study of dynamic software update safety restrictions. Technical Report CS-TR-4949, UMD, Department of Computer Science, 2009. http://www.cs.umd.edu/~hayden/papers/tr-dsutest.pdf.

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

[13] Kristis Makris and Rida Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[14] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *EuroSys 2007*, March 2007.

[15] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[16] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS*, October 2007.

[17] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[18] Shaya Potter and Jason Nieh. Autopod: Unscheduled system updates with zero data loss. In *ICAC 2005: Proceedings of the 2nd IEEE International Conference on Autonomic Computing*, pages 367–368, Seattle, WA, USA, 2005. IEEE Press.

[19] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *In Symposium on Operating Systems Principles*, pages 170–185, 1999.

[20] Jonathan M. Smith. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev.*, 22(3):28–40, 1988.

[21] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[22] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *PLDI*, 2009.

[23] Unsanity. Application Enhancer – enhance the applications by loading modules. http://www.unsanity.com/haxies/ape.

[24] vsftpd: Very secure ftp daemon. http://vsftpd.beasts.org/.