State Transfer for Clear and Efficient Runtime Upgrades

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), in which software is updated while it runs, 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 state transfer updating library for C/C++ programs that we are developing that seeks to redress the difficulties of in-place updating, and we report on our early experience updating VSFTPD using Ekiden. Our early experience suggests that state transfer provides the availability benefits of in-place DSU approaches and addresses 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 [3], [13], [7], [10], [16]. Ksplice [5] and Unsanity's Application Enhancer [17], 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 [4] touts 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 references to old code are redirected to the new. Such redirections were either anticipated by inserting levels of indirection (e.g., with a compiler, as in Ginseng [13], Erlang, and UpStare [11]), or forced on the running system via runtime program rewriting (as in Ksplice [5], POLUS [7], or Jvolve [16]). Patches also include *state transformation* code which runs after the patch is loaded, adjusting both data and control state.

While popular, in-place updating has several drawbacks: (1) tool support for updating is complex; (2) update compilers that use analyses that rely on static reasoning (e.g., alias analysis) may reject otherwise correct programs; (3) the mechanisms for updating often introduce steady-state runtime overhead; e.g., additional levels of indirection decrease performance by themselves and inhibit compiler optimization (4) reasoning about the behavior of an updated program places additional cognitive burden on the programmer; and (5) it requires extra effort to write the required state transformers. 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.

To address these drawbacks, we propose an alternative approach: *state transfer-based updating*. While state transfer updating is not new [8], we believe it deserves further study. We have been developing a library called Ekiden for state transfer-based updating for C and C++.[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 disadvantages of in-place updates: (1) no complicated tools are needed—state transfer updating can be implemented purely as a library that builds largely on existing infrastructure; (2) it has no steady-state overhead, and all compiler optimizations are fully available—the only overhead occurs at update time, a relatively rare event; (3) while the human effort of preparing a program is slightly increased, the cognitive burden is dramatically decreased, since we know exactly where and how updates may happen, increasing our confidence that they are correct; and (4) the effort in writing state transformers is reduced, since new state is automatically initialized by the new version of the program, and only old state that could be changed outside of initialization 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 support the redundancy necessary for this approach (e.g., operating

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

systems); and (4) more manual effort is required to insert annotations in 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 our design of Ekiden and report on preliminary experience using Ekiden to update VSFTPD, a server we have previously experimented with using Ginseng. We then compare Ekiden abstractly to in-place updating approaches. In our early experience, it was straightforward to implement state transfer for VSFTPD using the Ekiden library, 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 use a program with Ekiden, 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 may not need to be transferred if it will be correctly initialized during new-version start-up or can be derived from other state. In Figure 1, the state of nclients (representing the number of clients who have connected) is to be transferred, and so has been tagged using the function tag_st(), while const_val (which is only written during initialization) need not be transferred and was not tagged.

Update Points: An update is triggered when a dynamic patch becomes available and the program subsequently reaches a call to the Ekiden function update_point. Once an update is triggered, the new version is started, state is transferred (via a UNIX domain socket), and the old version is shut down.

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. In Figure 1, for example, we modified the program to jump directly to the handle_client loop if the incoming state indicates the update occurred at the update pointed labeled with "cl_loop." We expect this modification to be straightforward for most programs, but need more experience applying Ekiden to be sure.

```
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

Serialization: Any tagged state must be transferred between the old and new program. To do this requires that the state be properly serialized (for sending) and deserialized (on receipt). To specify how serialization should take place, the tagging function tag_st takes a deserializer function and a serializer function as arguments, in addition 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. In Figure 1, the nclients variable is deserialized when an update takes place, but is otherwise set to 0; when an update occurs, the value of nclients is serialized with export_int.

The functions export_int and import_int are part of an Ekiden library for serializing basic C types, and this API 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. While our experience is preliminary, thus far this tool makes writing serialization code quite easy.

Some kinds of program state, such as file descriptors and function pointers, are less straightforward to transfer. Open file descriptors are a particularly important type of program state that must be passed between versions to enable the new process to resume listening on open sockets and communicating with connected clients. Fortunately, because exec does not close open file descriptions, Ekiden can transfer open file descriptors as simple integers. Alternatively, file descriptors could be transferred using UNIX domain sockets and the sendmsg() system call [15]. Libraries such as the Ancillary Library [1] and Facebook's libafdt [2] simplify this process.

Since function addresses may not be consistent across program versions (indeed, some functions may not even persist across versions), they cannot be used for serialization. As a result, we currently require developers to write custom serializers for function pointers. Typically, this entails mapping function addresses to integers or strings and back. We hope to streamline this process in the future by adding function pointer annotations to our serialization generation tool.

B. Experience Updating vsftpd

To gain experience updating programs using Ekiden, we have constructed a sequence of four updatable versions of VSFTPD [18] (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.

Several minor changes to VSFTPD were required to configure the updating framework: installation of a UNIX signal handler to initiate an update upon receipt of a particular signal, saving the command-line arguments used to invoke the current version so they can be applied when starting the new version, and providing the file-system path to contain the updated executable.

We added two update points to VSFTPD, each to be reached between iterations of long-running loops. 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 directly enter 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 support direct "jumps" 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. These changes were similar to the conditional execution of handle_client added in Figure 1.

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 map from process IDs to connected IP addresses, a map from 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. 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. For VSFTPD, identifying and annotating state for transfer was easy. Future work will determine whether the same is true for programs that maintain more state.

Although there was not a large amount of state to transfer in VSFTPD, the code written to serialize and transform it accounted for the bulk of the modification effort. We hand-wrote serializers for two user-defined types in VSFTPD and found the process to be tedious and error prone, since serialization and deserialization functions are highly-coupled. This observation motivated development of our serialization-generation tool. The developer must still write the code to transform old program state to conform to the new version, but this effort is manageable since since most state was not changed between versions.

As we discussed earlier, function pointers can be problematic to transfer. An example we encountered in VSFTPD was a type-generic hash table that stores a function pointer to compute hash values. Because VSFTPD contains exactly two of these hash tables, each using a different function pointer for hashing, it was manageable to add handling for either function to the hash table serialization code. 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.

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 figure 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

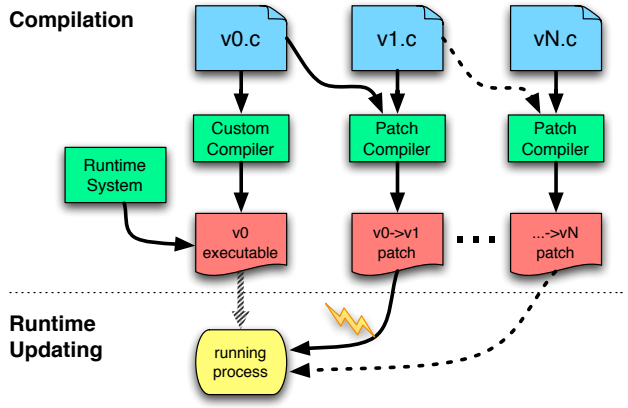

Fig. 2. In-place Update Compilation and Updating

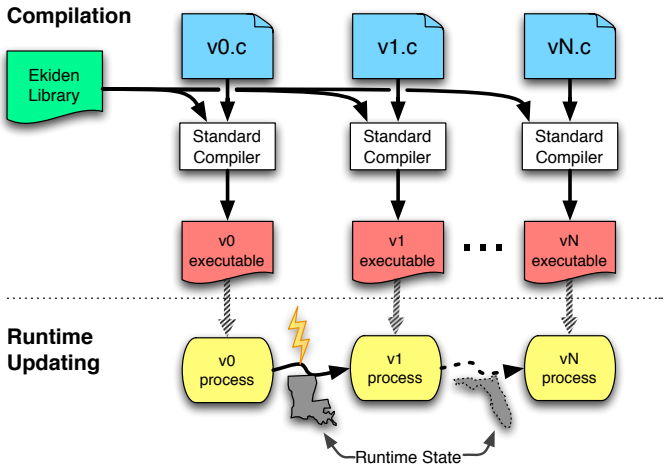

Fig. 3. Ekiden Compilation and Updating

analyses to support updating, e.g., as illustrated in Figure 2, which shows the updating process for Ginseng. In Ginseng, 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 updates require no such restriction. In our experience, non-standard compilation schemes and safety analyses are undesirable, 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 need not involve any non-standard tools, as illustrated in Figure 3. 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 so will not fail to handle programs that use odd or unsafe idioms.

Steady-state Performance: In-place updating schemes often introduce indirection so that function calls and variable accesses will reach the appropriate version. Adding indirection has two potential downsides: the indirection itself may incur a performance penalty, 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.

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 ocurred. 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 required 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 [9].

While both approaches require special code to support updating, the code for state transfer makes updating behavior explicit, while the changes for in-place DSU are for purposes that are not obvious in the source code, i.e., to support particular function changes in future updates.

An alternative in-place updating approach is *stack reconstruction*, where the active stack and program counter (PC) of the running program are updated to resume execution at the proper position in the new code. This is the approach used by UpStare [11] and DynAMOS [12]. Assuming that the stack and PC are updated correctly, this approach provides similar behavior to Ekiden's. In UpStare, loop bodies and code after loops need not be refactored into new functions, and following the update, the program will execute entirely using code from the new version. However, while stack reconstruction can provide greater update availability, inferring the required stack and PC mappings remains an open research question [6]. 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 pro-

gram state that has changed. We have identified two drawbacks with this requirement. First, bugfixes 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., tables of server commands) are written during initialization and are static thereafter. While changes to such data require manual state transformation under the in-place update approach, with state transfer-based updating they can be initialized 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 read-only state [13].

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.

Potter et al. [14] 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. [8] 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 evaluation, and we also need to develop techniques and tools that address challenging cases that may seem to favor in-place DSU techniques. These cases include updating programs that maintain a large state, transfering 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] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *USENIX Security*, 2005.

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

[14] 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.

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

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

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

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