Toward Practical Dynamic Software Updating

Iulian Neamtiu,‡ Gareth Stoyle,† Manuel Oriol,‡ and Michael Hicks‡
‡ University of Maryland † University of Cambridge

Abstract

Software systems are imperfect, so software updates are a fact of life. While typical software updates require stopping and restarting the program in question, many systems cannot afford to halt service, or would prefer not to. In this paper we argue that *dynamic software updating* (DSU), in which programs are updated while they run, provides superior flexibility when compared to standard alternatives. The challenge is in making DSU safe and easy to use; we argue that automated analyses are the key. We describe a prototype DSU implementation for C programs, and report our experience with some well-known open-source server programs (Apache, OpenSSH, `vsftpd`). For these programs, safe updates derived from standard releases were easy to generate, and performance was virtually unaffected by updateability.

1 Introduction

Many systems require continuous operation but nonetheless must be updated to fix bugs and add new features. For ISPs, credit card providers, brokerages, and on-line stores, being available 24/7 is synonymous with staying in business: an hour of downtime can cost hundreds of thousands, or even millions of dollars [9]. Many more systems would *prefer* on-line upgrades in lieu of having to stop and restart the system every time it must be patched; an obvious example is the personal operating system.

There are two basic approaches in common use. The first applies to *multi-tier* applications in which client requests are redirected by a stateless *load manager* to stateful nodes that implement the application logic [4, 10]. To upgrade node A, the balancer directs new requests away from A, the operator waits for A's sessions to complete, and finally patches A and reintegrates it into the service. The second approach is simpler: take down the service temporarily (a "planned outage") to upgrade it. Most users take the latter approach when rebooting their personal operating system to apply security patches. One study [7] found that 75% of nearly 6000 outages of high-availability applications were planned for hardware and software maintenance.

Both of these approaches are coarse-grained: upgrades occur on the order of whole programs, timed with respect to an external API. In this paper, we argue for a more fine-grained approach, which we call *dynamic software updating* (DSU): patch software with new code and data on-the-fly, while it runs. DSU has two advantages over common approaches:

There is no need for redundant hardware. While redundant hardware is often used to support fault tolerance and will thus be available in this case, there are many systems that typically do not require extra hardware. Examples are communications components (e.g., routers, firewalls, NAT translators, etc.), simple Internet servers, monitoring systems, personal operating systems, and embedded control systems. DSU allows programs to be updated in-place, so no additional hardware is required.

Application state can be preserved across an update. Without DSU, state must be managed explicitly by the application if it is to be visible in the upgraded version. In some cases, this is natural: if applications store critical state in the file system or a database, the state can be accessed by the upgraded version when it restarts. More generally, however, an application's logic might have to change to handle stateful upgrades

properly. For example, to preserve performance-critical soft-state, like file and translation caches, or to allow connection state to migrate between versions, would require modifying both the existing application to save its state and the new application to start by restoring it. Moreover, application writers must have a good idea in advance about what state to save, since future enhancements may be unknown. Because an upgrade changes the code and data layout, general-purpose approaches like check-pointing or migration among virtual machines [7] are not applicable. DSU does not require changing applications, and allows state to be transformed nearly arbitrarily for use by the new code, without prior knowledge of future changes.

While DSU is a very difficult problem in general, we argue that DSU is practical for many long-running applications. We present some evidence for this assertion in the form of a prototype implementation we have developed, and describe our experience dynamically upgrading several open-source Internet servers, including Apache, `opensshd`, and `vsftpd`.

2 Dynamic Software Updating

To make dynamic software updating practical, we must address three broad challenges:

1. *DSU should not require extensive changes to applications.* Moreover, while an application writer will anticipate that software will be upgraded, he should not have to know what the form of that update will be.

2. *DSU should restrict the form of dynamic patches as little as possible.* In particular, programmers should be allowed to change data representations, change function prototypes, reorganize subroutines, etc. as they normally would. All of these changes should be accommodated by DSU and expressible within dynamic patches.

3. *Dynamic patches should be neither hard to write nor hard to establish as correct.* Both the form and timing of dynamic patches must be correct. If a well-formed patch is applied at the wrong time, it can lead to incorrect application behavior, which is unacceptable. It should be straightforward to write correct patches even for large systems, in which a programmer will likely not understand the entire system.

Unfortunately, no current system addresses all of these challenges. Most early DSU approaches [5] provided little assurance of correctness and were fairly inflexible. More recently, some approaches better ensure correctness at the cost of limiting update flexibility [3, 11], or provide finer-grained updates at the cost of fewer correctness guarantees [1, 6], notably a lack of static type-safety. Few systems provide help in writing dynamic patches. Finally, no flexible approach supports C programs, meaning that a wide variety of non-stop applications cannot be supported without recoding.

We believe the key to addressing these challenges is *automation*. In particular, the compiler can modify a program to make it amenable to dynamic patches, and tools can be used to automatically construct dynamic patches and choose appropriate execution times at which to apply them. This way, programmers have the same ease-of-use as the load balancing approach while achieving the greater benefits of DSU. While the automated tools cannot be complete, as the problems of timing and patch generation are in general undecidable, we believe that in practice they can do a good job, as evidenced by preliminary results (Section 3).

2.1 Approach

We have been developing an approach for dynamically updating C programs. The general idea is illustrated in Figure 1. For the initial version of a program `V0.c`, the compiler (labeled AC for "Analysis and Compilation") generates an updateable executable (`P_0`), along with some prototype and analysis information (`V0.vd`). This information is fed along with the next version to AC which generates a dynamic patch to be applied to the running system. Patches are applied via dynamic linking, with some patch initialization code called at dynamic link-time.

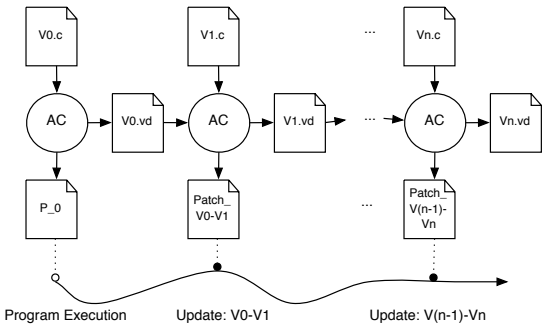

Figure 1: General Infrastructure.

To support dynamic patching, the compiler introduces a level of indirection for each function in the program; the patch initialization code replaces the targets to point to new versions it contains. In addition, *named types* (like `struct`s or `typedef`s) are compiled to contain a version field. Code that manipulates these types checks the version field, and, when necessary, a *type transformer* function is run to upgrade the representation. This function is provided with the dynamic patch.

Type transformers are generated automatically by comparing the old and new representations of types. For example, if the old version and new version of `struct ftp_session` differ only in the addition of a new field, then the generated type transformer will simply leave the other fields alone, and fill in a default for the new field. If a new representation grows beyond the size alloted for the old representation, we generate indirections to the new pieces, and compile the program to follow them. Surprisingly, we have found so far that programmers rarely change data representations in complex ways, and often do not change them at all, so our simple type transformers rarely need to be changed. State that is not stored in a named type must be transformed at dynamic link-time in the patch initialization code.

Dynamic patches must be applied at an appropriate time during the upgraded program's execution. This is the most challenging problem of DSU, because an ill-chosen moment could result in unpredictable behavior. Abstractly, correct timing depends on both the program and the update. In our system, C programs can contain explicit update points inserted by the programmer, which are meant to be independent of the form of the update. Many long-running applications are event-driven, and so determining reasonable update points is often not difficult: allow updates to occur in the event processing loop but not in its handlers. To further ensure that chosen update points are safe, the compiler uses a novel analysis to determine restrictions on changes to named types and function prototypes sufficient to ensure type safety [12].

For example, if the program contains an expression `x.f`, where `x` is a variable of type `struct T`, then placing an update point before dereferencing `x` is not safe because doing so might invalidate the expression `x.f`—what if the update removed the field `f` from the definition of `struct T`? The type-safe solution is then to place the update point after the expression `x.f`.

While type-safety does not imply correctness, the reverse is often the case. We believe the analysis can be extended to preserve other properties as well, including abstraction or encapsulation, which are also correlated with correctness.

Though there is much more to do to reach the goal of safe, flexible, non-intrusive DSU (Section 4), our current approach is sufficient to support a variety of real-world updates, as we show next.

3 Case Studies

We have applied our approach to three open source programs: `vsftpd`, `opensshd`, and Apache. We chose these programs because they are long-running, maintain soft state that could be usefully preserved across updates, and are in wide use. For each program we downloaded an older version, turned it into an upgradable program and ran it. Then we took a later version that implements critical updates, converted it into a dynamic patch, and upgraded the running software on-the-fly. In this section we describe our experience. Application characteristics and

Table 1: Stock vs. upgradable servers

Metrics	vsftpd		sshd		Apache	
	stock	*DSU*	*stock*	*DSU*	*stock*	*DSU*
Source LOC	13457	24471	36547	66093	44929	82332
Executable (KB)	131	217	336	485	332	539
% Overhead	-	1	-	3	-	2
Footprint (KB)	5260	5520	7448	8012	4754	4588

Operations			
Compilation (s)	4.5	17	22
Patch gen. (s)	3.2	8.7	13
Patch load (ns)	896	1027	2832
Patch size (KB)	32	46	170

performance[1] results are summarized in Table 1.

vsftpd The program `vsftpd`[2] stands for "Very Secure FTP Daemon." It has been carefully coded to avoid security vulnerabilities, like buffer overruns. We chose to upgrade `vsftpd` from version 2.0.1 to 2.0.2`pre2`. This version upgrade contains bug fixes for the server itself, plus workarounds to cope with buggy clients. In total, 21 functions changed, but no type was updated so no type transformer was needed.

To measure overhead, we stress-tested the upgradable server using the `dkftpbench` benchmark[3] plus our own scripts for transferring very large files. For a typical file size distribution and 0 to 800 simultaneous clients, `dkftpbench` showed no performance difference between the upgradable and stock server. Only for very large files was the upgradable server's transfer rate less, and here only by 1%.

OpenSSH We upgraded the `sshd` daemon[4] from version 3.7.1`p1` to 3.7.1`p2`. The new version contains critical bug fixes for problems such as double `free`, stack corruption and null pointer dereferences, plus around 20 'normal' bug fixes. In total, 19 functions changed, but again no type was updated.

We measured the upgradable server performance by transferring files of various size using `scp`. Again, for typical file sizes there was no visible difference, whereas for very large files the upgradable server's transfer rate was 3% less than the stock server.

Apache We upgraded the `Apache` web server[5] from version 1.3.24 to version 1.3.29 (thus a single patch contained multiple releases). This upgrade fixes lots of problems: 8 security issues (including 4 buffer overflows), a potential segfault, a potential server crash, 2 NULL pointer dereferences, 2 potential deadlocks, 2 potential endless loops and 58 other 'normal' bugs. In all, 129 functions and one `struct` type changed; the auto-generated type transformer required no modification.

We stress-tested the upgradable server using the `httperf` [8] load generator plus our own scripts for transferring very large files. `httperf` showed similar results for the upgradable and non-upgradable servers (we issued 0...1000 requests/sec). When transferring very large files the stock server was 2% faster on average.

Some final notes: First, finding candidate update points was straightforward and the static analysis confirmed they were type-safe. We find it interesting that few updates required changes to type definitions; this drastically simplifies the updating process. As we gain more experience, we expect to better understand the importance of type changes.

Second, compilation, patch generation, and dynamic patch application are all reasonably fast.

Finally, while the instrumented source code is almost double the size of non-instrumented code, memory footprint and performance are relatively unaffected, mainly because the instrumented code can be easily optimized by the compiler.

4 Further Research

Upgrading programs by current means, such as stop-and-restart or load balancing, can be costly, inconvenient, and inflexible. Dynamic software updating offers the ability to redress these prob-

[1] We conducted our experiments on a dual Xeon@2GHz with 1GB of RAM running Fedora Linux.

[2] `http://vsftpd.beasts.org`

[3] `http://www.kegel.com/dkftpbench`

[4] `http://www.openssh.com`

[5] `http://www.apache.org/`

lems, but only if it can be made simple and reliable. The system we have presented uses automation and analysis to improve the reliability of a flexible system. While our system is a promising first step, there is much to be done, both to support a wide variety of realistic applications and to further improve guarantees of reliability. We conclude this paper with some sketches of future directions.

Concurrency While our approach has targeted sequential programs, non-stop software is often implemented with multiple threads. Multi-threaded programs complicate the timing problem: while an update to a type `struct T` in one thread may be deemed safe by the analysis, another thread may be manipulating the same data. To avoid statically analyzing thread interactions, we can allow update points to act like a barrier: all threads must synchronize at legal update points before the update can proceed. It is safe to update the entire program if it is safe to update it at all of the synchronized update points. Threads that do not manipulate an updated resource need not synchronize. Moreover, static analysis can be used to avoid the possibility of deadlock [2]. To reduce contention further, and to overcome the limitations of static analysis, we can dynamically track those functions that are using a particular type, and only impose synchronization on them.

Transactional Updates Although our system prevents updates that would compromise type safety, it does not prove that updates are semantically correct. To provide further assurances, other researchers have employed language-level transactions in conjunction with updates [3] to ensure proper timing. The benefits are two-fold. First, transactions communicate semantic information about the program, as they group together "units of work." Therefore, we should prevent an update within a transaction that would permit calling the old version of one function and then the updated version of another. Second, the ability to abort transactions improves update availability. In the multi-threaded case, we need not pause compliant threads as proposed above, but can rather abort non-compliant ones, to reach consensus more quickly. The key difficulty with transactions is that they require programmer assistance.

Acknowledgments We thank Peter Sewell and Gavin Bierman for their contributions to this work.

References

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

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[3] C. Boyapati, B. Liskov, S. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *ACM SIGPLAN Notices*, 38(11):403–417, Nov. 2003.

[4] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[5] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *The Journal of Systems and Software*, 14(2):111–128, Feb. 1991.

[6] A. Goldberg and D. Robson. *Smalltalk 80 - the Language and its Implementation*. Addison-Wesley, Reading, 1989.

[7] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the ACM Conference on Architectural support for programming languages and operating systems*, pages 211–223. ACM Press, 2004.

[8] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[9] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D. A. Patterson, and K. Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Comput.*, 51(2):100–107, 2002.

[10] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[11] C. A. N. Soules, J. Appavoo, K. Hui, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration. In *Proc. USENIX Annual Technical Conference*, June 2003.

[12] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proceedings of ACM Symposium on Principles of programming Languages*, pages 183–194, 2005.