Towards Standardized Benchmarks for Dynamic Software Updating Systems

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

***Abstract*—Dynamic Software Updating (DSU) has been an active topic of research for at least the last 30 years. However, despite many recent advances, DSU has yet to see widespread adoption and deployment in practice. In this paper, we review a slice of the history of DSU research to study how DSU for C has evolved over the last two decades. We examine the ways DSU systems are evaluated in the research literature. We identify several shortcomings of the evaluation criteria that have been used, and propose key improvements. We believe that using better evaluation criteria can guide DSU research to produce systems that will be more practical, flexible, and usable.**

I. Introduction

Research on dynamically updating running software has been actively underway for at least the past 30 years. Numerous dynamic software updating (DSU) systems have been developed, targeting applications [1], [2], [3], [4], [5], [6], [7], [8], [9] and operating systems [10], [11], [12], and ranging from new compilers, to runtime frameworks, to libraries. While these systems vary widely in their implementation, they all allow programs to be upgraded at runtime with minimal interruption to execution, while retaining valuable state. Research systems have progressed significantly since the beginnings of the field, and the pace of research in DSU is accelerating. Recently, the first DSU startup, Ksplice [11], was bought by industry titan Oracle, suggesting the potential of the technology.

However, with the exception of Ksplice's customers, DSU is almost entirely unused in real-world systems. While DSU variants are popular in Java development, these systems are primarily used for debugging, and can support limited subsets of updates [13], [14]. Also surprisingly, though many DSU systems [4], [5], [11] have been released as free software [15], the free software community has not adopted any of these systems nor derived any more practical variants. This situation raises the question of why adoption is not more widespread.

We believe that to move beyond building systems that only Ph.D. researchers in DSU can use, to building systems that everyday programmers can use, the DSU research community needs to rethink how it decides what properties are desirable for a DSU system to have.

In this paper, we make two main contributions toward this end. First, we survey two decades of research on DSU for user-space C programs to help understand how DSU research has reached its current state, and to recall lessons learned over that time. (Section II) Second, we examine the ways that DSU systems have been evaluated in the research literature, and recommend improvements to evaluation strategies to help lead toward more practical DSU systems. Specifically, we propose the creation of a standard benchmark suite, suggest further research into defining the problems of update availability and flexibility, and call for direct usability studies of DSU systems (Section III).

We focus our discussion on DSU systems for user-space C programs due to space constraints, but our recommendations apply to DSU for other languages (e.g., Java) as well.

II. History of DSU

While dynamic software updating has existed conceptually since (at least) 1976 [1], DSU has only recently been demonstrated on real-world systems. We begin our survey of DSU for user-space programs in 1991, when the first application of DSU to C was published.

A. The 1990's

PODUS [3], [16], the Procedure Oriented Dynamic Updating System, is the earliest system we have found that supports updating C programs. It is implemented on SunOS, and purports compatibility with other UNIX variants. The authors demonstrate updating an example program written in C. PODUS uses *binary rewriting* to effect updates. Binary rewriting is widely used in later updating systems, and involves writing to the code segment of a running program to redirect old function calls to new versions.

To attempt to ensure update safety, PODUS only applies an update once no functions to be updated are live on the call stack. This restriction is sufficient (but not necessary) to ensure updates are type-safe. That is, assuming both the old and new programs are themselves type correct, dynamically updating the first to the second will not introduce any type errors. Limiting updates to inactive functions enforces what we call *activeness safety*. Activeness safety has been adopted by many DSU systems, for C and other languages.

Possibly the most significant work on DSU in the 90's is Gupta's *On-line software version change using state transfer between processes* [2]. This system uses a novel mechanism

called *state transfer* to effect an update by transferring some of the state of the running program to a specially prepared variant of the subsequent version of that program, started as a separate process. However, Gupta's main contribution is a formal notion of *update validity*. To summarize, an update is *valid* if it transforms an existing program's state into a state that could be constructed by executing the new version of a program from the start. In later work [17], Gupta proves that update validity is undecidable in the general case. This publication is the most cited DSU work of the 1990's. Update systems in the 2000's typically included safety guarantees approximating validity. Overall, the 1990's were a formative period for DSU research. While active, most work in this decade was speculative rather than concrete.

B. The 2000's

The second millennium saw the rise of an empirical focus in DSU, as researchers began measuring performance and other characteristics of implementations. We will consider four systems from this decade: Opus, Ginseng, PoLUS, and UpStare.

OPUS [18] is a DSU system that targets security patches, rather than full program versions. As far as we are aware, OPUS is the first dynamic updating system that was applied to real-world C programs. It effects updates using binary rewriting, similarly to PODUS.

Ginseng [4], [19] is a DSU system that implements, for C, a mechanism formalized in earlier publications [20]. Ginseng was the first DSU system flexible enough to apply dynamic updates derived from sequences of actual releases of real-world programs. Unlike all previous update systems, Ginseng permits updating active functions on the stack, provided these functions pass a static type-safety analysis. Updates are only permitted at *update points* designated by the programmer in the source program, with the goal of simplifying reasoning about the update [21]. Recent versions support updating multi-threaded programs [19].

Following the literature [20], we refer to this safety mechanism as *con-freeness safety*, so-called because the static analysis ensures the code does not include concrete references to values whose type is changed by an update.

Ginseng makes a program dynamically updatable via source-to-source translation; thus, Ginseng is largely architecture-independent. We refer to Ginseng's code updating mechanism as *indirection insertion*: Among other changes, Ginseng's translation phase changes function calls to indirect through jump tables. Ginseng also wraps each access to a structured type with a function call. Once an update is triggered, these wrapper functions perform much of the required transformation of the program's data, in place. Thus, Ginseng performs state transformation *lazily*. To facilitate in-place state transformation, Ginseng adds "slop" to C **struct**s. This slop is overwritten when fields grow or are added to types.

PoLUS [5] focuses on updating programs that have no direct support for DSU, with special consideration for multi-threaded programs. PoLUS is able to apply updates immediately as they become available, unlike other systems that wait for safety constraints to be satisfied. PoLUS does not require any changes to programs to support updating. PoLUS's runtime system is implemented as a standalone program. This program attaches to target programs using ptrace and uses binary rewriting to effect an update. Since old and new code may be running simultaneously, PoLUS requires the developer to provide *bi-directional coercion functions*, which are used whenever old or new code accesses state from the other version. PoLUS's runtime framework forces the equivalent of a global lock on state access during an update, since all access to state is mediated by the bi-directional coercion functions. When all old-version threads have begun executing new code, all state is transformed to the new representation, and the old code is unloaded.

UpStare [6] uses a novel mechanism, *stack reconstruction* [22], to implement updates. This approach is similar to Gupta's state transfer, but effects the update within a single process rather than transferring state between processes. UpStare's compiler inserts instrumentation that enables it to unwind the stack to the lowest depth on update, and then rewind it to an equivalent stack in the new version of the program. UpStare inserts both update points and *continuation points* to facilitate updating. Continuation points are inserted before every function call and update point, and update points at the beginning of every loop and function. UpStare infers a mapping between continuation points that allows it to rewind the appropriate point to continue program execution.

UpStare collects all global variables into one **struct**, and all parameters and locals for functions into one **struct** per function. These **struct**s are then updated according to UpStare's customizable rules for updating types.

When an update is requested in a multi-threaded program, the first thread to reach an update point acquires a *coordinator lock*. Other threads then block on this lock. UpStare wraps calls to lock and unlock so that the coordinator thread can detect when every thread is either locked on the coordinator lock, or an application lock held by a thread locked on the coordinator lock. Once every thread is blocked, UpStare initiates an update.

C. The 2010's

While the decade is still young, the 2010's have seen two new DSU systems for C proposed at the time of this writing: Ekiden [8] and its successor, Kitsune [9].

Ekiden is implemented entirely as a library. It exports an API that developers can use to enable dynamic updating in their program. At run time, when an update becomes available, Ekiden forks a new process and begins executing the new version, which receives serialized state from the old

version to initialize the new version. This is a more portable implementation of Gupta's state transfer idea [2].

Kitsune is implemented as a runtime framework, a source-to-source translator, and a transformation code generator. Programs are compiled to shared libraries after being passed through Kitsune's translator. These shared libraries are loaded by Kitsune's runtime framework and executed. At update time, the runtime loads the shared library of the new version and begins executing at `main`.

Ekiden and Kitsune conceptually share the same process for updating state. In both, state to be updated is marked by the developer, and in the new version, state is initialized either as usual, or by using the value from the previous version. Ekiden and Kitsune also both allow updates of any form at any programmer-inserted update point. Type errors cannot occur since no old code is ever executed following an update.

Kitsune supports updating multi-threaded programs by making an update point act as a thread barrier once an update is requested. Ekiden does not support multi-threading.

III. Evaluating DSU Systems

As we have seen, DSU for C has evolved tremendously over the last 20 years. As a consequence, the research literature includes a wide range of benchmarks and metrics used to assess proposed systems. We believe that the time has come to standardize some aspects of the assessment process, to promote progress in the field.

Toward this end, we examine the programs to which DSU has been applied, and also examine three categories of metrics: *performance*, including the steady-state overhead of the updating mechanism; *flexibility*, including what kinds of updates are permitted; and *usability*, including the developer effort required to apply the DSU system.

A. Target Programs

Figure 1 summarizes the programs that have been used to evaluate PoLUS, Ginseng, UpStare, and Kitsune. OPUS target programs are not considered; the changes supported by OPUS are limited in scope to the small code changes typical of security patches. For each target program, the figure lists its lines of code, concurrency, and the range and number of versions supported by each system.

The data shown in this figure suggests that to date, it is quite difficult to compare different DSU systems, as there is little overlap in the programs each system is applied to. VSFTPD seems to be the only de facto standard program. Further, while six of the eleven programs are multi-threaded, Ginseng and Kitsune together account for most of the evaluation of these programs (five of the six).

For the DSU field to move toward practicality, we need more uniform benchmarks, including both the programs and the conditions under which they are run. We propose that the community develop a standard benchmark, composed of free software [15] programs and a series of relevant tests. The characteristics of a DSU benchmark remain to be determined, but at least we believe the included programs should be of various sizes; be both single- and multi-threaded; have variable (and varying) amounts of in-flight state; and exhibit changes of varying complexity.

In the remainder of this section we consider the metrics that could be used to evaluate these benchmark programs.

B. Performance

Performance measures in the literature generally fall into three categories: steady-state overhead, compilation overhead, and update availability. A standard benchmark for DSU should reliably measure all three.

Steady-state overhead: Most DSU systems introduce some overhead during normal program execution, and there is a general sense in the community that such overhead should be minimized. PoLUS introduces fairly small overhead, limited to one `jmp` instruction per indirected function. It also imposes the temporary overhead of calling the bidirectional coercion functions while the state is being updated. Ginseng introduces a larger overhead, due to its use of indirection and support of lazy state transformation. Makris et al. measured execution time overhead to be as high as 150% in the best case for the KissFFT program [6] (whose structure amplifies Ginseng's overheads). UpStare lies between PoLUS and Ginseng, inserting a large amount of indirection on function calls, but much less on data access. Further, Ginseng and UpStare's program transformation provoke slowdowns by increasing branch prediction difficulty and cache misses [6]. All steady-state overhead in Kitsune comes from the requirement to compile programs to relocatable shared objects, which may call functions through a lookup table. This can be eliminated on some platforms through linker flags. Ekiden has no steady-state overhead.

DSU systems may also induce memory overhead. However, memory overhead is rarely reported in the literature. Ginseng reports up to 40% larger memory footprint, due to its inserted "slop" space [4]. In the fastest-running case in KissFFT, UpStare reports a memory increase of 53.7% [6]. Ekiden and Kitsune induce no steady-state memory overhead, but potentially double memory requirements at update time. PoLUS and OPUS do not report memory overhead.

Clearly there are many different tradeoffs to be made in how much steady-state overhead is induced by a DSU system. Thus, a benchmark must include workloads that are sufficient to illuminate the differences. For example, benchmark programs must be run long enough for steady-state overhead to be measurable, and should include workloads representative of fielded systems.

Compilation Overhead: Most DSU systems involve a compilation step, which can be costly. For example, Ginseng performs a static analysis which invokes an external constraint solver. The authors reported that this analysis

Program	LoC	MT	PoLUS	Ginseng	UpStare	Kitsune
KissFFT	1,936				1.2.0 (1)	
Memcached	4,181	×		1.2.2–1.2.5 (4)		1.2.2–1.2.4 (3)
vsftpd	12,202		2.0.0–2.0.4 (5)	1.1.0–2.0.3 (13)	1.1.0–2.0.6 (14)	1.1.0–2.0.6 (14)
Redis	13,387	×				2.0.0–2.0.4 (5)
IceCast	15,759	×		2.2.0–2.3.1 (5)		2.2.0–2.3.1 (5)
Space Tyrant	20,223	×		0.307–0.351 (7)		
zebra	45,568			0.92a–0.95a (6)		
openssh	58,104		3.2.3–3.6 (5)	3.5p1–4.2p1 (11)		
Tor	76,090					0.2.1.18–0.2.1.30 (13)
apache	315,381	×	2.1.7–2.2.0 (4)			
PostGreSQL	369,000				7.4.16–7.4.17 (1)	

Figure 1. C programs used to evaluate DSU systems

dominated compilation time, which reached 100s in the worst case (for OpenSSH). This overhead might not scale to larger programs. Most systems do not report compilation time, making comparison difficult.

Availability: Many DSU systems include technology to improve update availability, typically measured as the time required to apply an update. Some authors believe minimizing this time is essential [6], [5], and make significant usability sacrifices to do so. For example, the authors of Kitsune manually added just six update points to VSFTPD [9], whereas UpStare automatically inserts 613 update points in the same program [6]. Thus, the UpStare user must consider many more update points when trying to decide if an update is correct. Update availability can be improved at the cost of steady-state overhead by shifting state transformation from being performed eagerly at update time to being performed lazily during subsequent execution.

Despite a strong focus in the literature on update availability, we have not seen a clear justification for why update availability is important, beyond the trivial case where an update never occurs. Update development time is measured in terms of person-hours, -days, or -weeks; it seems unlikely that differences on the order of seconds or even tens of seconds in update deployment would be critical. Additionally, many programs for which DSU would be useful provide services over the Internet. Clients of such programs already must cope with large communication delays and intermittent service; such clients may have a surprisingly large tolerance for pauses while updates are applied. Preliminary results from Hayden et al [23] indicate that even in multithreaded programs, programs with a small number of manually selected update points reach quiescence within reasonable bounds. Thus, unless a particular system has a clear demand for it, we think focusing on maximizing availability is likely a premature optimization.

The work necessary to construct a representative benchmark would help determine just how important update availability is. For example, a benchmark could include requirements, derived from the field, on update availability.

C. Flexibility

Many DSU systems have limitations on the set of allowed updates either by design or by quirk of implementation.

Code changes: While most code changes are supported in DSU systems, there are some corner cases that are not always handled. For example, many systems [5], [6] cannot update the `main`, because it is only called once.

Data changes: UpStare, Ekiden, and Kitsune support all possible changes to types. Ginseng conceptually supports all possible type changes, but in reality can only expand **`struct`**s until the added slop space has been filled. PoLUS also conceptually supports all possible type changes, but assumes that there is a bi-directional transformation function for all possible types, which may not be the case in practice. For example, consider an implementation of a set that is updated to implement a bag. It is unclear what the correct behavior of this program would be, if both old and new code were executing and manipulating the same state.

While we suspect that flexibility in updates is essential in practice, we lack convincing evidence of this. We believe the community needs to conduct a wide-ranging study of programs and their updates (beyond the small set of programs in Figure 1) to determine how much flexibility is needed. A standard benchmark suite should cover updates of various scales to accurately measure a system's flexibility.

D. Usability

While ease of use is an essential precursor to widespread DSU adoption, we are not aware of any direct studies of DSU system usability. Moreover, while DSU systems often include anecdotal reports on usability, they are typically based on the experience of the DSU system designer applying their tools, rather than an unbiased third-party.

Evaluations of DSU systems often use changed lines of code as a proxy for developer effort. However, this is problematic because some individual changes require a great deal more consideration than others. For example, Ginseng reports low numbers for changed lines of code [4]. However, many of these changes were to satisfy Ginseng's static analysis. Each change may correspond to great deal

of developer time spent comprehending the analysis, and eventually arriving on a single-line change that satisfies it. In contrast, many changes in Ekiden's modifications to VSFTPD come from annotating variables as updatable, which is simple to comprehend. The time involved in single-line changes between Ginseng and Ekiden are not equivalent, and the same is true of other systems. We need better metrics to facilitate direct comparisons between DSU systems.

Another barrier to DSU usability is the difficulty in reasoning about the behavior of a program under an update. We believe this is a function of how simple the behavior of an update system is, and how clear the behavior is to the developer. For example, we suspect that PoLUS and Ginseng are difficult to reason about, because immediately after an update, the program is in a complex state containing old and new code and data. In contrast, we believe Ekiden, Kitsune, and UpStare are easier to reason about, because following an update in any of these systems, only ancillary code or new version code is executed, and in a well-defined order. However, while our opinions are supported by our anecdotal experience, we do not know whether they are true.

We think the time is now ripe for direct usability studies of DSU systems. These could range from controlled user studies in the lab, during which participants are asked to make a particular program work with a given DSU system; to longitudinal studies in the field, in which researchers observe participants experiences with DSU over the long term. Such studies would provide invaluable insights into the essential characteristics that make DSU systems usable in practice, and could also shed light on other questions, such as how much performance and flexibility is required from a DSU system.

IV. Conclusion

Dynamic software updating systems have grown over the last two decades from toy languages and systems to powerful frameworks capable of updating large, enterprise-grade programs. We believe the time is ripe to move DSU research into practice, and to achieve this end, our community needs to focus on improving the way we evaluate performance, flexibility, and usability.

One common phrase in the authors' discussions while we were building Ekiden was "taking over the world." Taking over the world could mean seeing DSU integrated into existing update distribution systems. It could mean user interfaces being updated to reflect DSU—picture Firefox, upon downloading an update, giving the user the chance to "Apply changes on the fly." Ultimately, DSU could radically improve software by enhancing stability and security. If we believe what we write in our introductions about the cost of downtime due to upgrades, we in the DSU community should be committed to taking over the world!

Acknowledgments: This research was supported by the partnership between UMIACS and the Laboratory for Telecommunications Sciences and NSF grant CCF-0910530.

References

[1] R. S. Fabry, "How to design a system in which modules can be changed on the fly," in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE '76, 1976.

[2] D. Gupta and P. Jalote, "On-line software version change using state transfer between processes," *Software Practice and Experience*, vol. 23, no. 9.

[3] M. E. Segal, O. Frieder, O. Frieder, and M. E. Sega, "On dynamically updating a computer program: From concept to prototype," *Journal of Systems and Software*, vol. 14, 1991.

[4] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI*, 2006.

[5] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *ICSE*, 2007, pp. 271–281.

[6] K. Makris and R. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *USENIX ATC*, 2009.

[7] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A VM-centric approach," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2009.

[8] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime upgrades," in *HotSWUp*, 2011.

[9] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," Tech. Rep. UMD CS-TR-5008, 2012.

[10] K. Makris and K. D. Ryu, "Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels," in *EuroSys 2007*, March 2007.

[11] J. Arnold and M. F. Kaashoek, "Ksplice: automatic rebootless kernel updates," in *EuroSys*, 2009.

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

[13] "Jrebel," "http://zeroturnaround.com/jrebel/".

[14] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of java applications – balancing change flexibility vs programming transparency," *J. Softw. Maint. Evol.*, vol. 21, no. 2.

[15] R. M. Stallman, "What is free software?" https://www.gnu.org/philosophy/free-sw.html.

[16] M. Segal and O. Frieder, "On-the-fly program modification: systems for dynamic updating," *Software, IEEE*, vol. 10, no. 2, Mar 1993.

[17] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE TSE*, vol. 22, no. 2, 1996.

[18] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: online patches and updates for security," in *USENIX Security*, 2005.

[19] I. Neamtiu and M. Hicks, "Safe and timely dynamic updates for multi-threaded programs," in *PLDI*, Jun. 2009.

[20] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "*Mutatis Mutandis*: Safe and flexible dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 4, 2007.

[21] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, 2005.

[22] R. A. Bazzi, K. Makris, P. Nayeri, and J. Shen, "Dynamic Software Updates: The State Mapping Problem," in *The 2nd ACM Workshop on Hot Topics in Software Upgrades (HotSWUp '09)*, 2009.

[23] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster, "A study of dynamic software update quiescence in multi-threaded programs," 2012.