Report on the

Fourth Workshop on Hot Topics in Software Upgrades (HotSWUp 2012)

http://www.hotswup.org/2012/

Karla Saur
Dept. of Computer Science
University of Maryland
College Park, MD
ksaur@cs.umd.edu

Iulian Neamtiu
Dept. of Computer Science and Engineering
University of California, Riverside
Riverside, CA
neamtiu@cs.ucr.edu

ABSTRACT

The Fourth Workshop on Hot Topics in Software Upgrades (HotSWUp 2012) was held on June 3, 2012 in Zurich, Switzerland. The workshop was co-located with ICSE 2012. The goal of HotSWUp is to identify, through interdisciplinary collaboration, cutting-edge research ideas for implementing software upgrades. The workshop combined presentations of peer-reviewed research papers with a keynote speech on how empirical software engineering can help reduce update-induced failures. The audience included researchers and practitioners from academia and industry. In addition to the technical presentations, the program allowed ample time for discussions, which were driven by debate questions provided in advance by the presenters.

HotSWUp provides a premier forum for discussing problems that are often considered niche topics in the established research communities. For example, the technical discussions at HotSWUp'12 covered dynamic software updates, package management tools, using model-checking and verification to verify updates, empirical software engineering and repository mining, and highlighted many synergies among these and other topics.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.8 [**Software Engineering**]: Metrics—*complexity measures,Performance measures Process metrics,Product metrics*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Processors**]: Compilers; D.4.7 [**Operating Systems**]: Organization and Design; K.6.3 [**Management of Computing and Information Systems**]: Software Management

General Terms

Management, Experimentation, Human Factors, Performance, Reliability

Keywords

Software upgrades, dynamic software update, model checking, virtual machines, package management

HotSWUp 2012 Overview

The HotSWUp 2012 program featured a keynote address and three research sessions; each session was followed by a discussion period. The keynote, by Martin Pinzger, described how mining software repositories can help reduce update-induced failures by identifying failure-prone source files and binaries. The first session included work related to dynamic software updating; the second covered topics related to the safety of dynamic updates and software upgrades; and the final session included work on model-checking software changes and dynamic updates. This report summarizes the presentations and discussion that took place during each session.

Keynote Address

Software Repository Mining for Improving Software Upgrades by *Martin Pinzger, Delft University of Technology*

Martin started with the motivation for his talk: software upgrade error messages are frustrating and confusing. His research area is mining software repositories, so the natural connection to HotSWUp, and the guiding motive for his talk was "Can we mine software repositories to reduce upgrade failures?".

The first piece of work he presented was on detecting fine-grained source code changes, and using them to predict bugs [13]. Their findings suggest that using SCC and detecting changes at a fine-grained AST level performs better at predicting bugs compared to churn (lines modified). The next line of work focused on investigating whether developer-module networks can be used to predict failures [26]. To verify this, they use data collected from analyzing developer contributions to Windows Vista binaries. They set up a regression analysis where the independent variable was the number of pre-release commits to a binary, and the dependent variable was the number of post-release bugs in the binary. Their findings have indicated that central modules are more failure-prone than peripheral ones, and that centrality measures can predict 83% of failure-prone binaries in Windows Vista. As a consequence, their recommendation for central binaries was to increase testing effort, consider refactoring, and rethink how contributions and contributors are organized. These two lines of work indicate several promising ideas for research in software upgrades that are facilitated by mining repositories: identify upgrade-critical components, identify upgrade bugs, collect anecdotes of upgrade failures [20].

Session 1

How To Have your Cake and Eat It Too: Dynamic Software Updating with Just in Time Overhead by *Rida Bazzi, Bryan Topp, and Iulian Neamtiu* [4]

Rida Bazzi gave the talk. The main motivation of the work was espousing and trying to address the inherent tension in dynamic updates: on one hand, to permit dynamic updates program state must be available, but exposing the state imposes an inherent overhead because it prevents certain compile optimizations. Moreover, dynamic update (DSU) systems introduce additional overhead by periodically saving the state. For example, if the program contains update points, compiler optimizations might move code up or down around the update points, or optimize away certain variables that contain state necessary for update. More generally, optimizations that lead to functions being inlined, variables being eliminated, merged, or to instructions being reordered will pose potential problems for updates that rely on these variables. Hence, if the programmer needs to rely on consistent state in order to perform the update, the compiler will be denied certain optimization opportunities. The authors propose a solution to this tension called *switching gears:* during normal execution, a "high gear" version of the program runs; this version's only overhead is to save program state. When an update is signaled, the execution is switched to a "low gear" version that supports dynamic updates and has high overhead; the update is performed, and then execution is switched back to high gear. Preliminary performance experiments indicate that, for CPU-bound applications, this approach incurs only around 10% steady-state overhead, whereas prior approaches such as Ginseng [24] and UpStare [18] can incur 40% to 100% overhead on such applications.

Michael Hicks agreed that the argument [that the tension is inherent] is valid, but indicated having less-frequent update points as a way to minimize the impact on performance. Cristian Cadar asked what was the baseline when measuring the overhead of the switching gears approach: the baseline was the program with full compiler optimizations enabled.

A Study of Dynamic Software Update Quiescence for Multithreaded Programs by *Christopher Hayden, Karla Saur, Michael Hicks, and Jeffrey Foster* [15]

Karla gave the talk. Allowing updates to take place only at well-defined update points simplifies the reasoning of program correctness. However, with multithreaded programs all threads must hit an update point before an update can take place, so there is the potential risk of update points delaying an update for too long, even indefinitely in the case of deadlock. The authors conducted a study of 6 diverse multithreaded server programs to measure the amount of delay that occurs before all threads in a program are at an update point. They concluded that although some programs had no problems, some programs would not ever be ready for updating because of threads blocking due to condition variables or I/O. To solve this, the authors created a library, QBench, that allows blocking calls to be interrupted, enabling or expediting the time to the ability to perform an update. They concluded that with the QBench library, all programs they tested had all threads at update points (and therefore were ready for update) within 0.155 to 107.558 ms, and most were below 1 ms.

The first question asked was how the authors suggest that programmers find these blocking calls, and how can programmers know they are not missing blocking calls. The authors used manual analysis of source code to find the blocking calls (such as searching for condition variables), and the results show that there are no missing blocking calls. Another participating commented that it may be possible to abstract blocking calls away into a library and then link against that to streamline the removal of blocking calls. A final question was how does the time scale with the number of threads, e.g., 100 threads. The answer was that the time depends more on code and the number of blocking calls than number of threads; many threads automatically don't block and cause no delay.

Towards Standardized Benchmarks for Dynamic Software Updating Systems by *Edward K Smith, Michael Hicks and Jeffrey S Foster* [28]

Edward gave the talk. The motivation behind the work was the observation that DSU adoption in the "real world" lags behind, even though DSU research is booming. A brief history of DSU systems was presented, from PODUS (1989, 1991) [27, 12], to Erlang (1999) [2] to OPUS (2005) [1] to Ginseng (2006, 2009) [24, 22] to POLUS (2007) [8], to UpStare (2010) [18], Ekiden (2011) [16], and Kitsune (2012) [17]. The trend, as DSU has progressed with these systems, is to increase flexibility, and decrease programmer burden and performance overhead. However, a look at the state of practice indicates that DSU systems are not widely adopted. Therefore, the authors argue the need for benchmarks to compare relative advantages of DSU systems. In particular, they propose DSU system metrics such as performance (e.g., availability, steady-sate overhead), flexibility (changes supported), and usability (programmer burden). The talk closed with a call to the community to push for DSU adoption by providing DSU benchmarks.

Sander van der Burg asked if the authors have looked at component updates in addition to the source-level updates characteristic of the aforementioned systems. The answer was yes, e.g., POLUS performs updates at the binary level. Rida Bazzi suggested adding correctness as a metric, which was agreed on as a good point. Another question raised the point of Java DSU, where the application genre is quite different from the genre of C DSU applications prior work has most focused on. A final comment suggested that real-world programs (e.g., desktop applications) use a different programming model compared to the server programs traditionally used to test DSU.

Discussion

A discussion session followed the talks. The first question was "how do DSU system authors choose benchmarks?" Rida Bazzi replied that the choice process is biased: program understanding is an important factor, i.e., DSU researchers pick systems that are easy to understand and update.

The success of Ksplice—a DSU system for the Linux kernel [3]—was brought up. Ksplice is widely used in the industry so there is compelling evidence that people care about dynamic updates enough to buy a subscription. The question then, is what made Ksplice popular: "killer" apps (e.g., security patches) or ease of use (Ksplice is close to a "push-button" update model where constructing and applying updates is very easy)? Michael Hicks mentioned that Ksplice takes usability head-on (perhaps a reason for its success), because security bugs are easily fixed with, say, 3-line updates. However, performing whole-release updates using Ksplice is problematic at best, because the massive changes in whole-release updates make safety a serious issue. The conclusion was that perhaps both factors (usability and small patches being sufficient for security fixes) led to Ksplice's success.

Another participant pointed out Android apps: they must be ready to stop/restart so maybe the community should focus on that platform; an enabler might be the fact that Android app updates seem to be mostly additive (added code and features and very few deletions). Michael Wahler suggested that the researchers, especially academic ones, should talk to industry and understand what kind of upgrades and state transfer are required in a practical setting. Danny Dig mentioned that, based on his experience discussing with developers from industry, DSU systems cannot require burdensome or unusual annotations because developers in the industry are not likely to accept/embrace them. Martin Pinzger suggested that a static analysis approach, akin to FindBugs, that advises programmers of update-critical statements might prove effective.

Session 2

Safe and Automated State Transfer for Secure and Reliable Live Update by *Cristiano Giuffrida and Andrew S. Tanenbaum* [14]

Cristiano gave the talk. He presented an approach for live update that supports automated state transfer, including pointer transfer and dynamic object reallocation. The approach is based on non-intrusive compiler-based instrumentation and uses metadata to track (introspect) objects. Essentially, when reaching an update point after an update is signaled, a new process is instantiated and the state is transferred. The transfer is observed by a state transfer monitor. After the transfer is complete, control is transferred to the new process and old process is cleaned up. Objects are tracked by intercepting `malloc` (via a `malloc` wrapper), and all the usual data structure change idioms [21] are supported.

The first question was whether object copying time was an issue; as expected, this time depends on the number of objects in the executing application. The second question asked about applications that the authors tested their approach on; the answer was mainly two applications: a research operating system (MINIX), and a prototype for user programs.

Atomic Dynamic Upgrades Using Software Transactional Memory by *Luís Pina and João Cachopo* [25]

Luís gave the talk. Their work was motivated by the observation that immediate update semantics (after an update, objects are converted before the program resumes at the new version) is intuitive, while lazy semantics (objects are converted upon first access, which can be long after the control resumes [24]) is non-disruptive, as the time to convert objects to the new version is amortized. Hence their solution aims to provide immediate semantics safety and lazy semantics performance. A potential issue can arise when conversion code tries to access program state that has already been converted. To reason about safety, they use transactions—code in a transaction is atomic with respect to the update, i.e., it appears to execute at either the old version or the new version [23]. Their solution to this tension is to keep the old versions of objects in memory after the update, so that post-update transactions running at the the old version see the objects at the old version. Then, whenever a post-update transaction encounters an object that has not been converted yet, a concurrent conversion transaction is started and completed before the original transaction resumes. Their system is flexible, e.g., it supports class splitting/deletion. They implemented a prototype named DuST'M using a bytecode rewriting stage after compilation that enhances the original program with support for dynamic software upgrades. The resulting transformed program runs on top of any JVM. Therefore, they do not require any special JVM to run and update programs using DuST'M. Based on their evaluation, lazy upgrades are far less disruptive than immediate updates, though (naturally) slower than performing no updates.

The first question was if there were any limitations to their approach. Luís answered that one requirement was that the underlying system must support transactions, e.g., via an STM; without transactions, their scheme does not work (note that STM support imposes an overhead). The second question asked what instrumentation is needed to support the scheme; the answer was that every instance is accessed through a handle.

A Generic Approach for Deploying and Upgrading Mutable Software Components by *Sander van der Burg* [29]

The author started with the motivation: software deployment is getting more complicated, (operating) systems are getting bigger, software configurations are difficult to reproduce, and upgrading may break the system. Therefore, there is a need for deployment automation. In prior work they have developed Nix [10], a package manager supporting reliable, reproducible package deployment and atomic local upgrades. Disnix [31, 30] extends the Nix package manager to support reliable and reproducible deployment of service-oriented systems onto networks of machines. In their approach, packages are encoded with a hash code in the name representing the dependencies (architecture, libraries, build procedure). While Nix and Disnix can manage static components, mutable components must be handled separately, and this is performed by the *Dysnomia* extension, which is new for this work. Dysnomia requires that components implement five operations, activate, deactivate, snapshot, incremental-snapshot, and restore. The semantics of these operations is application-dependent, but it allows Dysnomia to store these mutable components in a container and support isolation and snapshots for reproducibility. A disadvantage of the current scheme is that creating snapshots prior to upgrades can take very long, which is problematic, e.g., if the component is a database that clients need access to. In order to safely upgrade a system, they must avoid any interference from end-users that might affect the dump phase. Therefore they block end-user access and always wait until the entire dump has been made. As an optimization, the blocking time window can be shortened by using incremental dumps, although it may still be expensive for upgrading systems under high load.

Petr Hosek asked about the number of users for the system—the speaker said the system has about 25 users and it was also running on his laptop. The next question was whether Sander had considered using Disnix (which incorporates Dysnomia) to deploy software in a university lab's network. The answer was that a different system, Charon is used in that setting. Charon's purpose is to do distributed infrastructure deployment and to take care of the systems component, whereas Disnix takes care of deploying the application components of service-oriented systems, (but it does not deploy the underlying system configurations, e.g. the operating systems and system services). The last question asked if the build process can be parallelized to run on multiple machines. The answer was yes; as Nix borrows concepts from purely functional programming languages and every build is defined as a function, Nix can easily determine the closures and determine which functions can be run in parallel.

Discussion

The first point of discussion was not specifically targeted at the speakers in the session, but rather more generally at identifying settings where dynamic or incremental update techniques are not worth applying. For example, what if dynamic update time is higher than the time it would take to restart? On a related note, what if the time to generate binary patches is longer than to sending the whole new binary? One participant offered that large in-memory state does not mean the whole state is important [and thus warranting dynamic updates to avoid losing the state]. An example is the caching daemon Memcached [19] which is widely used on Web servers to cache content so that clients can be served from memory and avoid going to the disk; it is undesirable to restart Memcached very often as performance is degraded while it rebuilds the state to load objects into the cache.

A question for Cristiano Giuffrida asked to clarify their definition or rollback; in their case, rollback means the program can recover from crashes, but can not recover form corrupt state. A second question was whether in their system the requirement for state transfer is inherent or is due to C's weak type system (put another way, if their approach was for Java, would it need annotations?). Cristiano answered that applying their technique would probably be easier to other languages especially if they support, e.g., variant types (unlike C which only has unions).

Cristiano then asked whether state annotation [to identify state that must be converted at update] is a burden or a blessing? The consensus was that one-time programmer intervention to identify such state (which tends to be needed in the beginning of the evolution) is acceptable. His second question was whether the audience strongly wants hot rollback (in case the update goes wrong); the answer was yes, e.g., if an error occurs when updating the graphic card drivers the system can become unusable. Finally, how should I/O be buffered to support rollback? A solution might be to use Dumitraş's staged upgrades [11].

Session 3

Verification of Software Changes with ExpliSAT by *Hana Chockler and Sitvanit Ruah* [9]

This paper was presented by Julia Rubin, a colleague of the authors. She began by explaining that model checking provides the benefit of verification but the process is very time-consuming and does not scale well to large programs. The authors built upon their prior work on ExpliSAT, which is a concolic model checker combining concrete and symbolic model checking. ExpliSAT works by traversing the control flow graph and creates all possible control paths. New in this paper is the path traversal heuristic: traverse paths that have changed first. Their system passes the changes from the new version to the 'goto-cc' compiler, which simplifies the program's control flow. Then, the simplified version is passed to ExpliSAT with the update, and that output is then passed to the SAT solver to validate the expressions. The current approach works for sequential programs, but does not work for concurrent programs (future work). They tested ExpliSAT on a C++ program used in the ITER EU Project's thermo-nuclear reactor robot that changes the divertor cassette. ExpliSAT did not terminate in several hours [while the system presented in their work found the bug in several seconds].

Michael Hicks noted that this work sounds like dynamic symbolic execution and asked how is this work different from dynamic symbolic execution. Cristian Cadar asked if the authors are actually running the program and if it was similar to C-Prover, the model checker. Julia responded that they are not actually running the program; it is a concrete traversal of the control graph. The paths are computed on-the-fly and in every traversal the tool takes a different path, which is what the authors mean by concolic.

Multi-Version Software Updates by *Cristian Cadar and Petr Hosek* [5]

Petr gave the talk. He began by explaining that there have been many critical reliability issues in software updates. For example, Lighttpd web server developers fixed a small bug which unfortunately also broke a different feature. For a period of 12 months, the users had to either stick with the old version or upgrade to a new buggy version. In fact, bug fixes have a 14%–24% chance of being buggy themselves [33]. Existing approaches such as verification and validation are helpful, but despite their usage, we still see bugs in real-world systems. The authors' goal is to improve the execution of upgraded software to provide the benefits of the newer version but the stability of the older version. In their approach, the authors run both the old and new version in parallel and coordinate the execution of the two versions. They use the output of the correctly executing version at any given time. The multi-version application is run on top of a virtulization framework alongside the conventional application. This ensures that the multi-version application acts as one program to the external world. The authors must resolve diversions by synchronizing between the two versions at multiple levels of abstraction (protocol level, system calls), and handling the diversions by using the output of the new version by default. They must balance the trade-off between ease of synchronization and stability. The authors implemented a prototype for x86 in which they run two versions of a program with small differences in behavior and they successfully survived a number of crash bugs in Redis. The authors plan to improve performance overhead and non-crashing divergences in the future.

Danny Dig noted that the authors are currently running whole copies of the program and wondered if it would be possible to slice it and only run the part of the program where we see the divergences; it would waste less CPU cycles and might be more lightweight, as right now the authors are running the same program several times. Petr responded that they realize their idea might not be the most energy-efficient, but their current process is not much different than fully loading the CPU. The problem is identifying the differences, forking the execution, and then trying to merge back—this is difficult due to changes in the runtime state. Cristian added that differences in runtime state exist, but if many different versions are in memory, sooner or later the problem will be that full instances must be run anyway. Danny followed up by asking if they could do state merging. The authors responded that they are not concerned with state transfer problems as long as the external behavior is the same.

Michael Hicks noted that in this approach, users are limited to changes that do not dramatically affect the semantics. If the behavior is a bit different, the versions could diverge in small ways. For example, the format of an output log message could change and use UNIX time instead of day-month-year. There will be a diversion every time but the user won't care. Petr responded that in many cases even when this happens, divergence can be ignored, but it is hard to determine which cases can be ignored.

Michael followed up by pointing out that in the case of a stateless change, if we change the way we internally represent the database, then behavior will diverge—some keys will be in new instance but not in old instance. Cristian explained that they cannot currently handle this; this is a current limitation. They are trying to infer these changes and there is some external research on this related to mapping the old state with the new state; they hope to build on this research. Danny asked what kind of changes they currently support, and Petr responded that they can handle changes to control flow and can cover multiple functions as long as the stack is the same. They cannot handle changes to the stack layout or data structure changes.

Theseus: whole updates of Java server applications by *Erwann Wernli* [32]

Erwann explained that the goal of his paper is to allow different versions of code to coexist by using contexts that isolate each version of the code. He proposed a user-defined updatability in which everything happens via objects, called Theseus. Theseus uses whole, lazy, or eventual migration and does not impose any timing constraints and uses the garbage collection to force transformations if necessary. Contexts are first-class: first instantiate the context and then invoke methods to trigger the update. In his approach, he used incremental rather than global updates, and must restart threads so that the installation completes. Erwann built a prototype an applied it to 5 versions of the web-server Jetty, which is bound by I/O. He performed a load test and didn't notice any obvious overhead.

Danny Dig asked how the shared objects are found, and if extra state is used for this purpose. Erwann explained that when objects are potentially shared, they are reachable by traversal. There is also the case of garbage collection in which migration must be forced: the traversal starts at the root of objects that are known to be shared. In his approach, there is no extra state, but rather he traverses the whole graph that he knows is reachable. When the program calls invoke on the new context, it does some new action on the new object.

Danny asked whether he handles Java 7 where the programmer is given threads that are no longer reachable. Erwann responded that Jetty was using a threadpool which was somewhat similar and the program was able to handle that.

Sander van der Burg asked that, since Theseus does not use proxies, how are instance IDs preserved among upgrades? Erwann responded that he keeps a synthetic object to use as the identity. Sander then asked that if I have one object that is referenced by two others, how do you deal with this in an upgrade? Erwann responded that on the switch to the new version, he tracks which are shared common to the two others; there is no proxy, but the two are linked together so they belong together.

Cristian Cadar asked about the two transform functions: one forward and one backward. How are these made? Erwann responded that the transform functions are completely generated by the programmer, but that he plans to automate this in the future.

Discussion

Sander van der Burg asked Wernli about coming from a user perspective: “how should potential users use your approach? Does it work by changing pieces of the code and then the update occurs automatically?” Wernli responded that the programmers must change their application to use the program and then do the reflective call in the application for the update. van der Burg added that from an end-user perspective, if more fine-grained changes are supported, it would be very beneficial, however then users must get acquainted with the Theseus approach and change their process to changing the code directly. Wernli stated that he focused more on updating the Jetty server itself (focused on the the middleware) and focused less on the end-user. At this point the clarification questions ended and the discussion segued into more general software upgrade topics.

On Overhead:
Petr Hosek asked a question related to the overhead problem. He explained that when they started working on their approach, they tried to run benchmarks and noticed that on CPU-bounded applications, even for example on Redis, the latency increase was less than 1 millisecond. So even if the overhead is a factor of, say, 3x, the increase to 2 or 3 milliseconds isn’t significant. What is an acceptable overhead to the user, and how to bound/measure this? Pina pointed out that latency is per user interaction, and bounded to load. i.e., it is not up to the users if they can tolerate the increasing latency, but it is up to the service provider to decide to tolerate lower throughput. Iulian Neamtiu added that, based on anecdotal experience, software vendors cannot tolerate overhead, even 10% is unacceptable. Cristian Cadar stated that it seems like 10% overhead should definitely be OK; in fact, users would accept a 2x–3x program overhead when they cannot tell the speed difference, but it is really annoying when programs crash. Neamtiu pointed out that perhaps the right question to ask is about energy consumption—if the battery lasts only 1/3 as long, then that is a problem. Explaining this performance overhead (e.g., for Cloud programs where cycles are paid for) to CFOs watching the bottom line, would be impossible. Related to Cristian and Petr’s work, Iulian mentioned that the authors could get some pointers from looking at the log and replay community regarding which system calls to replay and where to handle non-determinism. For example, there is no need to replay `getpid()` as that is irrelevant for high-level tasks. But on the other hand, `read()` or `write()` calls must be recorded. The log and replay community might provide some turnkey solutions. Finally, gdb extensions exist that support deterministic replay.

Relating Software Updating to Product Line:
Julia Rubin stated that she is from the product line of work, with families of related products for target customers. Her question was: do participants think that some technologies/techniques from software updating can be used in product line community? The main challenge is to maintain a set of products that are very similar but slightly different. Iulian Neamtiu stated that this is a question for the architecture part of software engineering—as a software architect you have your architecture/design right if no major redesign is necessary when adding a new product to the line. Danny Dig mentioned that he is interested in the notion of programs as first-class citizens; take the same transformation to a different branch. Wernli noted that this is similar to the case where programmers construct virtual classes and nest them. Julia said that based on her experience with product lines, these techniques are not widely used in industry, mostly just in academia. Sander van der Burg stated that he works with Philips Medical Systems, and in their product line, technology is used to reassemble medical imaging software. Dependency management is a problem; objects must be there at runtime and there are no means of statically verifying if they are all available. Sander said he has some partial solutions to the problem in his current deployment. He uses hashcodes to identify components in a unique manner and uses conservative garbage collection techniques to search for pointers and determine the required runtime dependencies. He mentioned that maybe we could use similar techniques in modeling product lines. Julia said that something for upgrades can be applied to the product line related problems, and we can forge new solutions.

Security Patches:
Michael Hicks stated that he served on a government panel, where he talked about static security patches. As it turns out, many security vulnerabilities that are exploited are ones that patches already exist for, but patches have not been applied. In other words, there are very few zero-day attacks; most attacks are from un-upgraded systems. The government agency organizing the panel would really like systems to get patched quickly, but more importantly patched *reliably*—they are very hesitant to take a patch from the internet and just apply it. An example of this is updating the Flash player: to perform the upgrade, they [the agency] must get the patch and bring it to all systems; it takes a lot of manual effort to obtain the correct the patch and distribute it broadly and quickly. If the researchers could make this patch application happen *immediately* with almost no user intervention, it would make systems much more secure and palatable. Hence dynamic upgrades, and well tested upgrades, are a very fruitful direction. This is a motivation to keep in mind as the HotSWUp community pushes ahead in its work. Cristian Cadar added that there are many places where reliability is not important, but performance is. Then there are times where people care most about reliability and will sacrifice energy consumption. An example of this is search engines which might launch a query redundantly, say, 3 times in parallel just to keep latency low, which puts extra load on the data center, but the search engine providers are willing to do this for performance reasons. Cristian mentioned that this also relates to multi-version execution to speed up software in various contexts. Iulian Neamtiu said that you can decrease energy consumption by reusing parts in the parallel version, rather than redoing the entire execution. Cristian said that determining when to do the update (ex: laptop unplugged) can also be useful and he assumes even for DSU it might make sense to restart the application if it is actually cheaper, but other times, the trade-offs make DSU worth it. Iulian mentioned that Candea has proposed the use of micro-reboots to keep state clean [7, 6].

What’s Next:
Michael Wahler posed, “Say I have the Flash plug-in. Look at the whole process of getting the update, applying the update, etc. What’s missing? Or, if you look at multi-

version updates, when does it start to get messy to deploy the upgrade? What tools exist for this?" Cristian said that a lot of this is under work. From a testing and validation perspective, people have started using incremental testing, focusing on the testing the changed parts only. For example, the keynote speech gave predictions that would be useful in telling if a function is likely to be buggy. However, Iulian added, changing the path analysis because of the regression testing is expensive, and we need to minimize resource consumption to speed up the process. Pina stated that it would be useful to track static sources with dynamic statistics. We know we lose 90% of time in 10% of the code, so we are really keen about having that 10% of code completely bulletproof; if we could somehow relate those two, it would be really nice to know where to invest. Iulian stated that an Eclipse plug-in, presented at ICSM'09, aggregates data acorss multiple executions form the entire test suite and tells developers exactly how much time is spent on a specific path/method; this information, of course, depends on the quality of coverage and how solid the regression test suite is. Danny Dig added that you could just use a profiler, but Iulian pointed out that a profiler only gives you *one* execution.

Acknowledgments

We thank the program chairs, Danny Dig and Michael Wahler. We also thank the program committee: Don Batory, Walter Cazzola, Danny Dig, Michael Hicks, Bo Nørregaard Jørgensen, Julia Lawall, Brice Morin, Iulian Neamtiu, Tien N Nguyen, Manuel Oriol, Michael Wahler, Robert J. Walker, Pen-Chung Yew.

Finally, we would like to thank the authors of submitted and invited papers for providing the excellent content of the program and for their enthusiastic participation in the workshop.

1. REFERENCES

[1] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. Opus: online patches and updates for security. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 287–302, Berkeley, CA, USA, 2005. USENIX Association.

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

[3] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 187–198, New York, NY, USA, 2009. ACM.

[4] R. Bazzi, B. Topp, and I. Neamtiu. How to have your cake and eat it too: Dynamic software updating with just in time overhead. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[5] C. Cadar and P. Hosek. Multi-version software updates. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[6] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive micro-reboots: A soft-state system case study. In *Performance Evaluation Journal*, 2003.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[8] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *ICSE*, pages 271–281, 2007.

[9] H. Chockler and S. Ruah. Verification of software changes with explisat. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[10] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 583–592, Washington, DC, USA, 2004. IEEE Computer Society.

[11] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 18:1–18:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[12] 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, 1991.

[13] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, New York, NY, USA, 2011. ACM.

[14] C. Giuffrida and A. S. Tanenbaum. Safe and automated state transfer for secure and reliable live update. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[15] C. Hayden, K. Saur, M. Hicks, and J. Foster. A study of dynamic software update quiescence for multithreaded programs. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[16] C. Hayden, E. Smith, M. Hicks, and J. Foster. State transfer for clear and efficient runtime upgrades. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[17] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Appilcations (OOPSLA)*, Oct. 2012.

[18] K. Makris and R. Bazzi. Multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[19] Memcached. `http://www.danga.com/memcached/`.

[20] P. Narasimhan. Failures & Downtime Incidents. http://www.cs.cmu.edu/ priya/downtime.html.

[21] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 1–5, May 2005.

[22] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44:13–24, June 2009.

[23] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 37–50, Jan. 2008.

[24] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[25] L. Pina and J. Cachopo. Atomic dynamic upgrades using software transactional memory. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[26] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12, New York, NY, USA, 2008. ACM.

[27] M. E. Segal and O. Frieder. Dynamic program updating: a software maintenance technique for minimizing software downtime. *Journal of Software Maintenance*, 1(1):59–79, Sept. 1989.

[28] E. Smith, M. Hicks, and J. Foster. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[29] S. van der Burg. A generic approach for deploying and upgrading mutable software components. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[30] S. van der Burg and E. Dolstra. Automated deployment of a heterogeneous service-oriented system. In *36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 183 –190, Sept. 2010.

[31] S. van der Burg, E. Dolstra, and M. de Jonge. Atomic upgrading of distributed systems. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, pages 8:1–8:5, New York, NY, USA, 2008. ACM.

[32] E. Wernli. Theseus: whole updates of Java server applications. In *HotSWUp '12: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Zurich, Switzerland, 2012.

[33] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.