Report on the

Second ACM Workshop on Hot Topics in Software Upgrades (HotSWUp'09)

http://www.hotswup.org/2009/

Tudor Dumitraş
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA
tudor@cmu.edu

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

Eli Tilevich
Department of Computer Science
Virginia Tech
Blacksburg,VA
tilevich@cs.vt.edu

ABSTRACT

The Second ACM SIGPLAN Workshop on Hot Topics in Software Upgrades (HotSWUp'09) was held on 25 October 2009 in Orlando, FL. The workshop was co-located with OOPSLA 2009 and was sponsored by ACM SIGPLAN. Twenty researchers and practitioners, from the programming languages, systems, software engineering and database communities, attended HotSWUp'09.

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 invited presentations from well-known experts and a keynote speech on the practical issues related to performing large-scale upgrades. The audience included researchers and practitioners from academia, the industry (Facebook, ABB, Oracle) and the open-source community (AppUpdater). 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'09 covered dynamic software updates, package management tools, database schema upgrades, upgrades of systems with real-time constraints, *etc.*, and highlighted many synergies among these topics. Perhaps more interestingly, the industry presentations provided real-world examples of systems that a have strong requirement for online upgrades. These examples emphasized the magnitude of the software upgrade problems that the industry is facing today.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.7 [**Operating Systems**]: Organization and Design; H.2.1 [**Database Management**]: Logical 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, package management, database schema evolution, real-time upgrades

Keynote Address

Release Management And Software Deployment At Facebook by *David Reiss (Facebook)*

David Reiss explained that Facebook performs more upgrades than any other organization in the world. This is the result of Facebook's agile development culture, which compels the engineers to "move fast and break stuff." The defects introduced in this manner are fixed in the next upgrade cycle (or earlier).

Facebook upgrades are executed in weekly deployments. On each Tuesday, after two days of testing when ≈1,000 developers use the new features internally, the upgrade is deployed to the live site. The developers are responsible for the upgrade and its outcomes, rather than transferring the code to a department of professional system administrators. This approach ensures that the engineers assume responsibility for the development, testing and deployment of their software and that they "feel the pain" of the users. All the experimental code is kept in the main development trunk. New features are deployed by copying the changes to the live branch (i.e., via `svn copy`) and by pushing the code to all the front-end servers. At this point, Danny Dig remarked that code refactorings often result in breaking changes (see Dig et al. [7]). Reiss replied that refactorings are rare at Facebook. Moreover, conflicts do not occur often because the code does not diverge too much during one week.

Reiss then described three upgrade techniques used at Facebook. He first focused on an essential piece of the company's agile development environment, called Gatekeeper. Gatekeeper is a system developed in-house, that allows programmers to check whether a feature is enabled in the current version of the code. This prevents separate development branches and complex code merges; Facebook engineers update the code from the the development trunk and use Gatekeeper to keep the new features disabled until they are ready to be released. This effectively decouples the deployment of a new feature from its launch. Gatekeeper leads to inelegant code, peppered with if-ladders that check for the availability of various feature combinations. However, Facebook developers have found this programming style to be very effective, because the boundaries of different features and versions are reflected in the code, instead of being hidden in the branches of a revision control repository. Michael Hicks asked if Facebook eventually removes the Gatekeeper checks, to avoid performance degradation. Reiss explained that they have automated scripts that look for features that are enabled for 100% of the users and that have not changed in a while, and the developers try to remove the corresponding checks. However, there are checks (e.g., for enabling optimizations specific to the IA64 architecture) that are never removed.

The second technique that Reiss described targets a reduction of bandwidth usage when pushing upgrades to the front-end servers. Because the development machines and the production servers are usually located in different data centers, upgrades often require excessive bandwidth, even when sending code deltas only (e.g., through `rsync`). After evaluating several alternatives, Facebook decided to use BitTorrent to disseminate these updates. Reiss also described a technique for taking over the active TCP sockets of a server, by opening a UNIX-domain socket between the old and new versions and duplicating the socket of the active connection (i.e., using `dup2` and `libafdt` [1]).

These techniques were developed because Facebook favors online upgrades and tries to avoid downtime. To achieve this goal, the site provides a relaxed consistency guarantee: when you change your data you will see the effects immediately, but other users might experience a delay. Such inconsistencies might result from changes in either the application front-end or the storage back-end.

Back-end changes are infrequent and do not have a major impact on Facebook. Data migrations always require special handling code and usually render the site unavailable for the users affected by the upgrade (e.g., because `ALTER TABLE` is a very inefficient operation in MySQL). However, Facebook is able to avoid such migrations most of the time. The changes to database schemas are usually limited to adding columns and tables, and schema inconsistencies between the application and the database do not constitute a significant challenge for Facebook[1] (but see the presentation on "Automating Database Schema Evolution in Information System Upgrades," in the third session, for examples of systems where this is a major problem).

Front-end changes, however, have a major impact. A front-end upgrade is a long-running operation because many servers must be upgraded. During a front-end upgrade, the site is in a mixed-version mode: some servers run the new version while others, which have not been upgraded yet, continue to run the old version. It is therefore possible for a client to load the new version of the page (including JavaScript code), from a server that has been upgraded, and to invoke an AJAX callback that arrives at a server still running the old version of the front-end code. In this case, the server might be unable to process the callback because it corresponds to features introduced in the new version. These front-end inconsistencies represent a major challenge for Facebook.[2] Therefore, Reiss's debate question, addressed to the audience, was "How can we provide a consistent JavaScript environment to a browser that interacts with two or more versions of our back-end?"

This remained an open question.

Session 2: Upgrade Models And Mechanisms

Cooperative Update: A New Model for Dependable Live Update by *Cristiano Giuffrida and Andrew S. Tanenbaum (Vrije Universiteit, Amsterdam, The Netherlands)* [9]

After the keynote speech describing the state of the practice at Facebook, Cristiano Giuffrida took a step back and discussed the models for upgrading a system online, focusing on their dependability guarantees. To this end, Giuffrida referenced Michael Hicks's Ph.D. thesis [12], which describes two update models: the interrupt model and the invoke model. In the interrupt (push) model, the system is interrupted at an arbitrary point in time to apply the update and the state is transferred into the new version. Updating tools that follow the interrupt model (e.g., Ginseng [14]) enforce safety guarantees at runtime and are suitable for providing backward compatibility at the binary level. In the invoke (pull) model, the system reaches a valid update point and notifies the updating tool. In this case, the safety guarantees can be enforced through static or dynamic analysis, which are suitable for providing backward compatibility at the source level. Giuffrida questioned the scalability of these models in the presence of complex upgrades; for instance, open-source projects double in size every 14 months and require large amounts of code to be changed during an upgrade. This increases the complexity of state transfers (not only to handle data type changes), the probability that an update requires

[1]As Bobby Johnson, Facebook's Director of Engineering, explained in his OOPSLA keynote four days later, this is the result of the site's highly-connected user base. Because the friendship connections evolve continuously and do not produce stable clusters, the Facebook system scales better through horizontal partitioning (e.g., split users across several databases) than vertical partitioning (e.g., split the names and addresses in different database tables). This allows Facebook to avoid major schema changes.

[2]According to Johnson, this is, currently, Facebook's biggest problem.

an unbounded time to complete (when enforcing safety constraints eagerly), and the effort to inspect the code manually (to determine safe update points and to ensure the correctness of execution for all possible update states).

These two models aim to be transparent to the system developers, separating the development and upgrading concerns with the goal of supporting legacy code. Instead, Giuffrida proposed a cooperative update model, which trades transparency and backward compatibility for an increased reliability of the update process. Because the average time between updates is 30 days in highly-available systems, Giuffrida argued that providing relaxed reliability guarantees during a live update can have severe consequences. In the cooperative model, the system is receptive to changes and interprets the update properties to prepare for a live update. Updates are packaged along with metadata that describes the nature of the update, and an update manager translates this specification into an update protocol that is appropriate for the changes implemented. Giuffrida described a six-step procedure for performing a cooperative update. He explained that this live-update procedure is deterministic and bounded in time.

At the end of the talk, Alan Choi asked if system components must know the state of the other components they communicate with, during a cooperative update. Giuffrida replied that components must have sufficient information to avoid deadlock, but need not know the global state of the system.

Dynamic Software Updates for Real-Time Systems by *Michael Wahler, Stefan Richter (ABB Corporate Research, Switzerland) and Manuel Oriol (University of York, UK)* [15]

Michael Wahler remarked that, while the previous talks addressed systems that are updated weekly or monthly, in embedded real-time systems the average time between updates is one year. These systems have typical lifetimes of 20–30 years. The main concern in this case is meeting real-time deadlines. This is challenging, because, while much of the previous research focused on modifying the internal structures of the operating system to implement live update mechanisms, ABB uses commercial operating systems that cannot be altered.

An embedded system includes processing units that control physical elements (e.g., a power network), relying on sensors (e.g., an ammeter) and actuators (e.g., a circuit breaker). The operating systems available off-the-shelf for these embedded processors provide mechanisms that allowed ABB to build a component-based framework supporting dynamic updates. These mechanisms include remote debugging, message-passing interfaces and separate address spaces for threads.

The components execute cyclically on the processor, according to a static schedule that is established offline. The update mechanism uses the slack available in each scheduling cycle to perform the code changes and the state transfer required. Dig asked what kind of changes are supported by this framework, and Wahler gave bug fixes and changes in the communication protocol as examples. He also presented experimental results, illustrating the update of a component with 4 kB of state during the 2 ms of slack from a 5 ms scheduling cycle.

Wahler concluded by outlining several directions for future work, which include updating multiple components atomically and tolerating malicious behavior. However, he cautioned the audience that, in the embedded world, there is a lot of skepticism about whether live updates will ever become acceptable for customers with strict certification requirements.

On Performance of Delegation in Java by *Sebastian Götz (Dresden University of Technology, Germany) and Mario Pukall (Otto-von-Guericke University Magdeburg, Germany)* [10]

Sebastian Götz presented an experimental evaluation of the overhead imposed by delegation (dispatching a call to another method) in Java. In addition to being one of the key mechanisms of object-oriented design patterns, delegation is also used by many dynamic software updating mechanisms (e.g., function indirection) and contributes to their overhead.

The authors evaluated delegation chains of 1000 invocations, using 10 Java Virtual Machines (JVMs) on 3 operating systems, installed on 2 different machines. The results included the surprising observation that, in some cases, delegation improves performance by up to 8%. Götz explained that this happens because the just-in-time (JIT) compiler optimizes the execution on-the-fly. The more information the JIT compiler has, the better it gets at optimizing the program, for instance through inlining or by composing the delegated calls automatically. In other experiments, however, the performance penalty recorded was up to 50%.

Hicks questioned the choice of evaluating delegation chains of 1000 invocations. He explained that the chains might have 100 calls but never 1000, and the benefits observed might be exacerbated by these long delegation chains.

Eli Tilevich, the session chair, concluded the formal presentations from the first session by recalling the old parable that every problem in computer science can be solved by adding a level of indirection and every performance problem can be addressed by removing a level of indirection. He then remarked that the last talk seems to suggest that this is not always the case, because of optimizing JVMs.

Discussion

Giuffrida kick-started the discussion with the observation that the average bug lifetime in operating systems is 1.8 years, and, even for security patches, studies suggest a time-to-update of at least 10 days. He then suggested that the main focus of upgrading mechanisms should be the minimization of overhead, rather than the live update timing. Reiss replied that this is true only if the upgrade doesn't require downtime, and Hicks added that downtime shouldn't concern Facebook if it is below 50 ms (the threshold for human perception).

Tudor Dumitraş asked how can we bound the worst-case-execution-time (WCET) of state transfer, which seems to be necessary for the first two approaches presented in this session. Giuffrida clarified that his approach is not bounded in time, but guaranteed to terminate eventually. Wahler explained that if the transfer is short enough, it doesn't matter whether it has an exponential complexity. However, he conceded that there are no general guidelines for bounding the state transfer. Austin Anderson asked Götz why delegation seems to improve performance in some cases but not in others. Tilevich expanded on this by saying that benchmarking Java is tricky. For example, the extensive DaCapo benchmark suite [4] focuses on macro-benchmarks (real world applications with non-trivial memory loads). Tilevich wondered what can be learned from micro-benchmarks, such as the ones presented in this session. Götz replied that the goal of the paper was to assess the worst-case penalty due to delegation.

Giuffrida closed the discussion with the controversial statement that, unlike other research communities (e.g., grid computing), we have failed to debate the level of transparency that is appropriate for upgrading mechanisms.[3] Hicks commented that being transparent to legacy code is not a major concern for Facebook. He then explained that we aimed for transparency and focused on updating existing systems, which were designed without any concerns for software upgrades, to demonstrate the feasibility of online upgrades. Once this point was made, we can think about scalability, how to achieve efficiency, etc. Dig added that we have conducted some empirical studies in the past, but we need more. In particular, empirical studies should be done from the perspective of online upgrading—and not only software evolution—in order to determine what the most frequent changes are, and what changes are amenable to automation.

Session 3: OS And Database Upgrades

Online Application Upgrade Using Edition-Based Redefinition by *Alan Choi (Oracle Corporation, USA)* [5] (invited paper)

Alan Choi started his talk by presenting Oracle's business case for online upgrades: many customers demand this functionality. The necessity of online upgrades arises in various domains. Applications such as electrical-utility management systems, support systems for global companies (e.g., customer-relationship management), assembly-line manufacturing, e-commerce or online banking require 24/7 database availability. While continuous hardware availability can be solved by redundancy, we have no complete solution for continuous *software* availability. To mitigate this, Oracle 11g Release 2 allows online application upgrades using a hot rollover technique, called edition-based redefinition. This technique uses two separate editioning views: old clients use the old edition, new clients use the new edition. When data changes in either edition, cross-edition triggers (backward and forward) ensure that changes are propagated between the two views. Eventually, the old edition is retired.

[3]However, the chairs recall a lively debate, on the acceptable amount of programmer annotations for dynamic software updating techniques, held after the first session of HotSWUp'08.

This technique allows clients to test a new version before deploying it widely. Choi also presented a case study demonstrating the use of edition-based redefinition to support the upgrade of a human-resources support application.

Carlo Curino asked if there are any limitations on the form of the backward/forward triggers, *e.g.*, do they have to be invertible? Choi responded that trigger code is left entirely up to the application programmer. Hence, it is incumbent upon the programmer to write triggers that ensure that data is consistent between the old and new applications. Dumitraş asked what happens if a trigger is buggy, and Choi replied that triggers must be tested to make sure they function correctly. Dumitraş followed up his question by wondering why are back triggers necessary at all. Choi explained that, in some cases, Oracle customers require multiple versions to be active during the upgrade in order to switch to the new version gradually.

An Implementation of the Linux Software Repository Model for other Operating Systems by *Neil McNab (Appupdater Project, USA) and Anthony Bryan (Metalink Project, USA)* [13]

Neil McNab described Appupdater, an open-source system for "detecting, downloading and installing upgrades automatically." Appupdater currently works on Windows, but takes its inspiration from Linux. In particular, most Linux distributions use shared repositories and local package-management tools that make it easy to detect installed software and upgrade. However, in Windows and Mac OS this process is not streamlined, and applications (*e.g.*, Firefox, Adobe Reader) include custom-built upgrade mechanisms.

The design of a general-purpose upgrade mechanism needs to overcome two challenges:

- *Software Detection:* how can we detect what software or version is installed on the local machine? In Windows, some programs use the registry, others embed version information in the executable itself. Appupdater uses a local installed-packages database that stores the hash values of installed programs, and maps hash values to particular versions of applications.
- *Software Download:* how can we ensure the integrity of the installer we are attempting to download? The solution is to use Metalink, an open standard for representing metadata for downloader programs. In particular, Metalink descriptions can contain information about mirror lists, file hashes or file sizes.

Appupdater is open source and written in Python. It tries to ensure privacy by not allowing the repository owner to find out which software is installed locally, on the users' machines. McNab also discussed the future of the project, which includes porting Appupdater to Mac OS.

The audience asked what happens when somebody manages to add a malicious application to the application repository. McNab responded that we must trust the application providers. When asked if an Appupdater repository can become a possible target for a DNS poisoning attack, McNab acknowledged that DNS attacks are one of their concerns. A follow-up question considered the scenario where the local hash information on installed software is leaked, which could make the client susceptible to "probing." McNab acknowledged that local hashes must be closely guarded. In theory, they should only be communicated to the Appupdater repository, and not to other parties.

Automating Database Schema Evolution in Information System Upgrades by *Carlo Curino (Massachusetts Institute of Technology, USA), Hyun J. Moon (NEC Labs, USA) and Carlo Zaniolo (University of California, Los Angeles, USA)* [6]
(invited paper)

While the first talk in this session focused on mechanisms for supporting software upgrades that require database changes, Curino wondered what are the most common changes. In particular, the presentation focused on two problems: schema evolution and data evolution.

An analysis of Wikipedia shows that schemas do change a lot. Moreover, each evolution step can impact up to 70% of the queries issued by the application. Unfortunately, the common practice for dealing with schema evolution is manual schema migration. Ideally, migration would be automatic and would provide safety guarantees (*e.g.*, preserving integrity constraints). Curino presented an approach, based on Schema Modification Operators (SMOs), that automatically rewrites queries for 97% of Wikipedia's evolution steps. The remaining 2.8% are challenging because the user exploits knowledge about the application that uses the database. Sometimes the authors' auto-migration system is slow when compared to hand-crafted migrations that make use of this domain knowledge.

The second line of work—data evolution and database archival—centers around the archiving and querying of historic contents. The authors propose archiving using the original schema and automatically rewriting older or newer queries so they can work on the original schema.

At this point, the audience was wondering what general lessons can be learned from Curino's results, which only cover the evolution of Wikipedia. For instance, is the query-failure rate (after each evolution step) representative? Rida Bazzi and Iulian Neamtiu shared their work-in-progress observations that there is quite a large variation across applications. Curino agreed that more studies are needed. He explained that, using support from the NSF, his group is currently developing a benchmark for schema evolution, which will include hundreds of applications (`http://schemaevolution.org/`).

When asked what percentage of schema evolutions are invertible, Curino answered that, usually, schema changes do not lose data; for instance, `DELETE TABLE`, or `DELETE COLUMN` are rare. Hicks added that one can use lenses [8] to preserve back-and-forth invertibility. Dig pointed out that there is a parallel between schema evolution and refactorings; for example, we can see a relational databases as an object instance, and a query as the interface supported by that instance. Many of the techniques developed, over the past 20 years, for refactoring programs might apply for automating schema changes.

A member of the audience then asked if these results are representative for schema evolution in the corporate world (*e.g.*, how often do they use aggregates or stored procedures, which are difficult to handle automatically?). Curino acknowledged that more interaction with the industry would be beneficial to determine whether the results obtained from open-source systems, such as Wikipedia, are relevant for commercial systems. Choi said that, in his experience, almost all schema evolution is currently a manual process. Dumitraş asked what kinds of schema transformation impose downtime, and Reiss answered that, because Facebook can not tolerate downtime, their solution is to restrict the form of updates instead.

Discussion

As the debate following Curino's presentation segued into the discussion part of the session, McNab asked the audience one final question: is there a method to ensure that malicious software doesn't make it into the software repository of Appupdater? On solution, suggested by Reiss, was for Appupdater to vouch that the only software that is offered comes from trusted vendors.

Session 4: Dynamic Software Updating

Dynamic Software Updates: The State Mapping Problem by *Rida A. Bazzi, Kristis Makris, Peyman Nayeri and Jun Shen (Arizona State University, USA)* [3]
(invited paper)

Rida Bazzi focused on the following two issues of Dynamic Software Updating (DSU): (i) what mechanism should we use to perform state mapping? and (ii) how can we determine whether an update is safe? He described UpStare, a general mechanism for immediate updates to multi-threaded applications that works by redirecting mid-function code in old functions to mid-function code in the new function. UpStare uses stack reconstruction to update active functions. Blocking calls are transformed into non-blocking calls, and all threads block before an update.

The main contribution of the paper is about taming the state-mapping problem. The proposed solution assumes some degree of compatibility between versions; some differences are ignored to reduce the amount of state to be mapped. For example, Bazzi explained the concept of lightweight functions, which are guaranteed to exit in a bounded period of time. In other words, these functions have no loops, no recursion, no input, no synchronization, *etc.* The strategy is to start with lightweight functions and continue to heavyweight functions. UpStare waits for lightweight functions to exit and performs semantic checking with modifications.

During the presentation, the audience raised several questions. Hicks asked about a precise definition for heavyweight functions. Reiss pointed out that network applications cannot be free of heavy functions—there is always a heavy function if the program reads from a network socket. Hicks also raised the possibility of having multiple threads that run light weight functions and never exit. He asked whether the threads would have to be blocked in such cases. Bazzi suggested that these scenarios tend to be corner cases, as determined by the performance analysis of realistic programs.

Log functions are those functions that are heavily used. The presented approach can detect the "top" log function, but will miss some of them. Bazzi pointed out that around 10% of changes are due to automatically detected log functions. Hicks suggested that it may be beneficial to simply ignore calls to log functions. To ensure backward compatibility, the presented approach compares the semantics of old and new versions; components are then forced to have compatible semantics. For bug fixes, state mapping is either impossible or trivial. One solution is to checkpoint frequently. Hicks asked whether the purpose of checkpointing was to check whether state corruption occurs or not. In the interest of staying on time, the presenter skipped to the conclusions at end of the presentation.

Migrating Protocols In Multi-Threaded Message-Passing Systems by *Austin Anderson and Julian Rathke (University of Southampton, UK)* [2]

While much of existing DSU work is focused on type safety, Anderson's presentation focused on higher level properties. Specifically, this work is about ensuring high-level update safety in message passing systems. Anderson's approach assumes that there is no shared state. The authors use session typing for specifying communication protocols formally.

As an example, consider P1 sending two integers to P2. P2 responds with one integer. Updating P1 and P2 separately could cause an error if P1 and P2 are running different versions of the protocol and trying to exchange data. Therefore, the authors propose a solution based on update coordination using "runs", i.e., the two processes can be updated if P1 has finished a run and is withholding sending any more data to P2. This way, P1 can be updated while P2 consumes data at the old version; eventually, P2 will be updated. Safety is enforced through static analysis. The proposed solution can guarantee that each individual thread will eventually run the new protocol. A limitation of this approach is that a sender cannot be a receiver as well.

Reiss asked about the type of code that is amenable to this static analysis, and Anderson explained that he focused on a "lambda calculus-like" functional language. Choi wondered whether the approach assumes that each sender must have one receiver, and Anderson replied that one-to-many correspondences are supported. The final question concerned the accuracy of the approach in the presence of multiple instances. For example, what will happen when two instances of P1 are talking to P2? Anderson explained that, because all the messages are annotated with the receiver's information, the approach still works as expected.

Efficient Systematic Testing for Dynamically Updatable Software by *Christopher M. Hayden, Eric A. Hardisty, Michael Hicks and Jeffrey S. Foster (University of Maryland, College Park, USA)* [11]
(invited paper)

Chris Hayden presented the idea of verifying DSU through testing. A test minimization procedure is used. The problem is that DSU creates a new sources of errors, *e.g.*, if the update is applied at the wrong time. The authors use several safety check procedures to detect when updates can be applied; a standard safety check is Con-freeness Safety (CFS), *i.e.*, an update to a type T can be applied if no instances of T are in the current program point's continuation. If an update is not safe at the current program point, this is called a conflict. Safety checks only ensure some update safety properties, hence we want to use testing as a means of further verifying the correctness of updates.

The testing process works as follows: instrument the application to trace possible update points, and then try to apply the update at each point and see whether the test fails. In practice, however this approach is prohibitive because of numerous update points. Therefore, the authors observed that applying the update at many different points will yield equivalent results, so they perform update test minimization by detecting (via static analysis) the equivalence classes induced by update points, and only testing the update at one update point in each class. The authors have implemented their approach on top of Ginseng [14] and ran experiments with 11 versions of OpenSSH and 9 versions of Vsftpd. Their results show that update test minimization is very effective, and was able top achieve an 86–98% reduction in the number of update tests.

Dig asked how the authors managed to find conflicts automatically. The answer was that their safety notion was based on trace equivalence: if a program trace is unchanged when a patch is applied, *i.e.*, the patch does not conflict with the trace, then applying the patch is deemed safe. The audience asked whether this update safety model would work with protocol updates. The answer was that, since OpenSSH and Vsftpd use processes rather than threads, each process completes its portion of the sends and receives before it can update. Dig asked if the approach would still work properly in the presence of ping-pong patterns and shared state, and the authors conceded that this is one of the outstanding challenges for their approach. The audience asked if sending messages to yourself could help, but the presenter pointed out that it made no sense—you already have this data.

Iulian Neamtiu took one step back and asked why DSU systems use the Activeness Safety check (*i.e.*, prohibit changes to active code or data) at all if it does not ensure correctness. Michael Hicks replied that they really do not use it directly, but they modify it in some way.

Discussion

Hicks further suggested that systems should be built with dynamic updates in mind. Then it will be safer to make assumptions about what kind of changes can be made. Dig suggested that safety in this case really depends on the nature of a given application and its requirements. David Reiss pointed out a fundamental difference between the telephony and modern Web applications. While phones use long transactions, Web applications like Facebook deal with short transactions, which are usually at most several seconds long.

Acknowledgments

We thank the Program Committee: Gustavo Alonso, Taweesup Apiwattanapong, Umesh Bellur, Gavin Bierman, Danny Dig, Manuel Oriol, Mark E. Segal, Liuba Shrira. Thanks are also due to ACM for sponsoring the workshop, to Steve Marney, OOPSLA'09 Workshop Chair, to Chandra Krintz, SIGPLAN Vice-Chair, and to Adrienne Griscti, Program Coordinator of SIG Publications, for their precious advice and support. 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] LIBrary for Asynchronous File Descriptor Transfer. `http://sourceforge.net/projects/libafdt/`.

[2] A. Anderson and J. Rathke. Migrating protocols in multi-threaded message-passing systems. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[3] R. A. Bazzi, K. Makris, P. Nayeri, and J. Shen. Dynamic software updates: the state mapping problem. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[4] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[5] A. Choi. Online application upgrade using edition-based redefinition. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[6] C. Curino, H. J. Moon, and C. Zaniolo. Automating database schema evolution in information system upgrades. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[7] D. Dig and R. Johnson. How do APIs evolve&quest; a story of refactoring: Research articles. *Journal of Software Maintenance and Evolution*, 18(2):83–107, 2006.

[8] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming*, pages 383–396, Victoria, BC, Canada, 2008.

[9] C. Giuffrida and A. S. Tanenbaum. Cooperative update: a new model for dependable live update. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[10] S. Götz and M. Pukall. On performance of delegation in Java. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[11] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient systematic testing for dynamically updatable software. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[12] M. W. Hicks. *Dynamic Software Updating.* PhD thesis, The University of Pennsylvania, August 2001.

[13] N. McNab and A. Bryan. An implementation of the linux software repository model for other operating systems. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.

[14] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, Ottawa, Ontario, Canada, 2006.

[15] M. Wahler, S. Richter, and M. Oriol. Dynamic software updates for real-time systems. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, Orlando, Florida, 2009.