Multi-version Software Updates

Cristian Cadar Petr Hosek
Department of Computing
Imperial College London
{*c.cadar, p.hosek*}*@imperial.ac.uk*

***Abstract*—Software updates present a difficult challenge to the software maintenance process. Too often, updates result in failures, and users face the uncomfortable choice between using an old stable version which misses recent features and bug fixes, and using a new version which improves the software in certain ways, only to introduce other bugs and security vulnerabilities.**

In this position paper, we propose a radically new approach for performing software updates: whenever a new update becomes available, instead of upgrading the software to the new version, we instead run the new version in parallel with the old one. By carefully coordinating their executions and selecting the behavior of the more reliable version when they diverge, we can preserve the stability of the old version without giving up the features and bug fixes added to the new version.

We are currently focusing on a prototype system targeting multicore CPUs, but we believe this approach could also be deployed on other parallel platforms, such as GPGPUs and cloud environments.

***Keywords*-software updates, multi-version execution**

I. Introduction

Software updates are an integral part of the software maintenance process, with new software versions being released on a continuous basis. Unfortunately, software updates often result in failures, which makes many users reluctant to incorporate the latest patches made available by developers. As a result, users rely instead on outdated versions, which despite their relative stability, miss recent features and bug fixes and may be plagued by security vulnerabilities.

In this position paper, we propose a novel way of performing software updates, which aims to resolve the uncomfortable choice that users often face between using an old stable version which misses recent features and bug fixes, and using a new version which improves the software in certain ways, only to introduce other bugs and security vulnerabilities.

Our idea is simple yet effective: whenever a new update becomes available, instead of upgrading the software to the new version, we instead run the new version in parallel with the old one. By carefully coordinating their executions and selecting the behavior of the more reliable version when they diverge, we can preserve the stability of the old version without giving up the features and bug fixes added to the new version.

We believe that multi-version software updates are a timely solution in the context of today's computing platforms [4]. In recent years, we have witnessed the emergence of new technology—ranging from multicore processors to large-scale data centers—which provide an abundance of computational resources and a high degree of parallelism. Furthermore, these resources are often left idle, in which state they consume a large fraction of their full-utilization energy levels. For example, recent studies [1] have shown that servers in a data center usually operate at between 10% and 50% of their maximum utilization; however, even an energy efficient server consumes half its full power when doing virtually no work—and for regular servers, this ratio is much worse.

Our approach aims to improve the software update process by taking advantage of idle resources—such as idle cores on a CPU and idle servers in a data center—to run multiple versions of an application in parallel, with the goal of improving the overall reliability and security of the software being upgraded. Our current focus is on multicore processors, but we believe this solution could be adapted to work on other parallel platforms as well. Furthermore, this update mechanism could be extended to work with a large number of versions running in parallel and configured to balance conflicting requirements such as performance, reliability and energy consumption.

The rest of this paper is organized as follows. Section II motivates our approach using several real scenarios involving *Chrome*, *Vim*, *lighttpd* and *vsftpd*. Then, Section III gives an overview of our approach and Section IV discusses the possible ways in which it could be implemented, including an initial prototype that we developed for multicore platforms. Finally, Section V discusses related work and Section VI concludes.

II. Motivating Scenarios

In this section we motivate our approach using existing scenarios involving the *Chrome* browser and the *Vim* editor, as well as the *lighttpd* and *vsftpd* servers. These correspond to two categories of applications that could benefit from our multi-version software update approach: desktop applications such as web browsers and office tools for which reliability is a key concern; and network servers, with stringent security and dependability requirements.

Google Chrome[1] is one of the most widely used web browsers. Even though *Chrome* releases are tested extensively before deployment, they sometimes introduce new bugs that affect the stability of the browser. A concrete example is version 6.0.466.0, which introduced a bug that caused *Chrome* to crash when trying to load certain web pages over SSL.[2] One might argue that in this case the user should downgrade to an older version and wait until the bug is fixed. However, versions immediately preceding 6.0.466.0 suffer from a different bug,[3] which was introduced in version 6.0.438.0 and which crashes Chrome during certain sequences of repeated back and forward navigation.

Vim[4] is arguably one of the most popular editors under UNIX. In version 7.1.127, while trying to fix a memory leak, *Vim* developers introduced a double `free` bug that caused *Vim* to crash whenever the user tried to use a path completion feature. This bug made its way into *Ubuntu* 8.04, affecting a large number of users.[5]

lighttpd[6] is a popular web-server used by several high-traffic websites such as YouTube, Wikipedia, and Meebo. Despite its popularity, faulty updates are still a common occurrence in *lighttpd*. As one example, a patch introduced in April 2009[7] (correctly) fixed the way HTTP ETags are computed. Unfortunately, this fix broke the support for compression, which relied on the previous way in which ETags were computed and resulted in a segmentation fault whenever a client requested HTTP compression. This issue was only diagnosed and reported in March 2010[8] and fixed at the end of April 2010,[9] more than one year after it was introduced, leaving the server vulnerable to possible attacks in between.

vsftpd[10] is a fast and secure FTP server for UNIX systems. Version 2.2.0 added several new features such as network isolation, but unfortunately also introduced a bug[11] which triggered a segmentation fault whenever a client used the passive FTP mode. This bug made *vsftpd* practically unusable since the passive mode is being frequently used by clients behind firewalls. Despite being reported several times, this bug was only fixed in version 2.2.1, released more than two months after the bug was introduced.

All the scenarios presented above describe software updates which, while trying to add new features or bug fixes, also introduced new bugs that caused the code to crash under certain conditions. Improving the reliability of such updates is the main goal of our proposed approach: running both the old and the new version in parallel after an update can enable applications to survive more errors, without giving up the new features introduced by the update.

[1]http://www.google.com/chrome
[2]http://code.google.com/p/chromium/issues/detail?id=49197
[3]http://code.google.com/p/chromium/issues/detail?id=49721
[4]http://www.vim.org/
[5]https://bugs.launchpad.net/ubuntu/+source/vim/+bug/219546
[6]http://www.lighttpd.net/
[7]http://redmine.lighttpd.net/projects/lighttpd/repository/revisions/2438
[8]http://redmine.lighttpd.net/issues/2169
[9]http://redmine.lighttpd.net/projects/lighttpd/repository/revisions/2723
[10]https://security.appspot.com/vsftpd.html
[11]https://bugs.launchpad.net/ubuntu/+source/vsftpd/+bug/462749

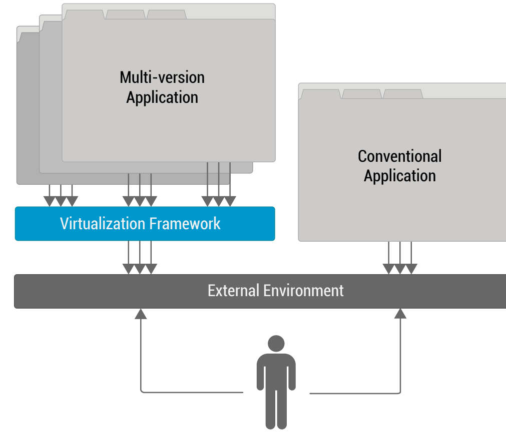

Figure 1. A platform running conventional and multi-version applications side by side.

III. Overview and Main Challenges

The main goal of our approach is to run multiple versions of an application in parallel, and synchronize their executions so that (1) users are given the illusion that they interact with a single version of the application, (2) the multi-version application is at least as reliable and secure as any of the individual versions in isolation, and (3) the synchronization mechanism incurs a reasonable performance overhead.

As shown in Figure 1, our solution requires a form of virtualization framework that would coordinate the execution of multiple application versions, and mediate their interaction with the external environment. Various such frameworks have been designed in the past in the context of running multiple automatically generated variants of a given application in parallel [3], [9], [18], and many of the techniques proposed in prior work can be reused in our context. To be practical, this coordination mechanism has to incur a reasonable overhead on top of native execution and ensure that the overall system is able to scale up and down the number of software versions run in parallel in order to balance conflicting requirements such as performance, reliability, security and energy consumption.

One particular challenge for our approach is to detect any divergences between different software versions, resolve them in such a way as to increase the overall reliability of the application, and finally synchronize again the different versions after their executions reconverge to the same behavior. Of course, we also need to handle the case in which the executions of different versions fail to reconverge to the same behavior after sufficient time.

Selecting the "correct" behavior of an application when different versions disagree is of course not possible in the general case without having access to a high-level specification. However, one could (1) focus on universal correctness properties, such as the absence of crashes, and (2) use various heuristics such as majority voting and favoring the latest application versions. For example, our current prototype resolves a divergence by always using the behavior of the version that has not crashed, and favoring the behavior of the latest version in all other cases. In this way, we ensure that the overall application has strictly fewer crashes than any of the individual versions, while still using the new features implemented in the latest version.

Note that one key aspect on which our approach relies is *having the different versions be alive at all times*. This ensures that applications can survive crashes that occur at different points in different versions, but adds the extra challenge of restarting crashed versions.

Finally, our approach needs a deployment strategy to decide what versions are run in parallel when the number of available resources (e.g., idle CPU cores) is limited. We envision several options—such as keeping the last n released versions (where n is the number of available resources), or keeping several very old stable versions—but the exact strategy should be decided on a case-by-case basis.

IV. Implementation Options and Current Prototype System

Our multi-version execution update mechanism can be implemented at multiple levels of abstraction. The simplest, but least flexible approach is to synchronize the different versions at the level of application inputs and outputs. This is particularly suitable for applications that use well-defined protocols such as web servers and web browsers. The main advantage of this option is that it can tolerate large differences between different software versions (in fact, one could even run different implementations, as in [24]), but the main downside is that it is not applicable in the general case, when the input-output specification is not available.

A more flexible approach, which we adopted in our prototype implementation, is to synchronize the different versions at the level of system calls. System calls represent the *external behavior* of an application, as they are the only way in which an application can interact with the surrounding environment. While this option allows fewer changes between different application versions (basically, the external behavior has to stay the same, although various optimizations are possible), it does not require any a priori knowledge of the application's input/output behavior, and is particularly suitable in the context of software updates, where the external application behavior usually remains unchanged. In fact, an empirical study that we conducted on several real applications (*Vim*, *lighttpd*, *redis*) has shown that while the source code of these applications changes on a regular basis, the external behavior changes only infrequently, often remaining stable as the software evolves.

An option similar to the system call interposition approach described above is to synchronize versions at the level of library calls. This has a performance advantage, as library functions can be executed only once, propagating the result to all versions. More generally, one can trade off the amount of code that is run (and synchronized) in parallel with the performance overhead achieved by the overall system. For example, one sensible option would be to avoid replicating the parts of the code that are highly trusted, e.g., those that have not been changed in the last several years, or those that can be statically proven to be safe.

We have implemented our approach in a prototype system targeted at multicore processors running Linux, using the system call interposition approach.[12] Currently, our prototype fully works with only two application versions, but we are working on adding support for a larger number of versions. As discussed in Section III, our prototype focuses on minimizing the number of crashes introduced via software updates, by using the behavior of the surviving version when one of the versions crashes, and the behavior of the latest version in all other cases. One important aspect of our prototype is its ability to restart a crashed application, which is accomplished by a combination of lightweight OS-level checkpointing and a form of runtime code patching which uses information from a binary static analysis pass.

Our prototype supports a wide range of Linux applications, in many cases with a reasonable performance overhead (21.48% on average for SPEC CINT CPU2006, but up to 17x on some other benchmarks). More importantly, we were able to use our prototype to survive a series of crash bugs in several real applications, such as *Coreutils*, *lighttpd*, and *redis*. However, our prototype still has a series of important limitations, which open up many opportunities for future development. We discuss some of them below.

Currently, our prototype intercepts every system call performed by each application version. This has two main disadvantages: first, any changes in system call behavior are flagged as a divergence.[13] We plan to improve this by using an epoch-based approach [22], in which multiple system calls are processed at a time, and their overall equivalence determined in a more flexible way (e.g., two sequences of writes can be considered to be equivalent as long as their overall result is the same).

Second, intercepting every system call often has an important impact on performance. We plan to alleviate this problem by using two techniques inspired by existing virtualization technology: (1) we plan to provide multi-version applications with a "paravirtualization" API [23]

[12] An earlier version of our prototype is described in [13].

[13] We optimize this by ignoring several system calls that can be safely replayed from the last checkpoint (e.g., `geteuid`), but the general problem remains.

that would allow them to communicate directly with the virtualization framework; and (2) we intend to combine this API with a binary translation approach [20], that would enable us to dynamically replace certain system calls with more efficient calls into the virtualization framework. The binary translation could also be used to dynamically replace code components that are proven to be safe and do not need to be replicated across multiple versions.

Furthermore, we plan to enhance our prototype implementation with the ability to dynamically adjust the number of versions that are run concurrently. This will ensure that multi-version applications will be able to consume all available resources (i.e., idle processor time) without affecting the overall system performance during peak load.

Finally, we would like to be able to transparently run multi-version applications on multiple underlying platforms, ranging from multicore processors to large-scale data centers. This requires the ability to span our virtualized environment across multiple logical as well as physical nodes. In particular, we aim to include the possibility of executing certain versions of an application remotely, to enable scenarios with hundreds or even thousands of application versions.

V. Related Work

The idea of concurrently running multiple versions (or a *multi-version execution*) of the same application was first explored in the context of N-version programming, a software development approach introduced in the 1970s in which multiple teams of programmers independently develop functionally equivalent versions of the same program in order to minimize the risk of having the same bugs in all versions [7]. During runtime, these versions are executed in parallel and majority voting is used to continue in the best possible way when a divergence occurs.

Recently, several researchers have proposed techniques that apply a form of N-version programming based on *automatically generated software variants* [3], [6], [9], [11], [18], [21], [24], [25]. For example, DieHard [3] uses heap over-provisioning and full randomization of object placement and memory reuse to run multiple replicas and reduce the likelihood that a memory error will have any effect; and Orchestra [18] runs two versions of the same application with stacks growing in opposite directions, and synchronizes their execution at the level of system calls, raising an alarm if any divergence is detected, which would have been triggered by a stack-based buffer overflow attack.

Cox et al. [9] propose a general framework for increasing application security by running in parallel several automatically generated diversified variants of the same program. The technique was implemented in two prototypes, one which runs the variants on different machines, and one which runs them on the same machine and synchronizes them at the system call level, using a modified Linux kernel.

There are two key differences between our approach and previous work in this space. First, we do not rely on automatically generated variants, but instead propose to use existing software versions as a mechanism for improving software updates. This also means that as opposed to previous solutions, the versions run in parallel are not semantically equivalent—this eliminates the challenge of generating diversified variants and creates opportunities in terms of recovery from failures, but also introduces additional challenges in terms of synchronizing the execution of the different versions. Second, while previous work has focused on detecting divergences, our key concern is to *survive* them, in order to increase the reliability, availability, and security of the overall application.

Research on surviving software failures has received a lot of attention in the past [5], [8], [15]–[17], [19], [22], and our proposed approach can benefit from the techniques developed in this context.

Previous work on improving the software update process has looked at different aspects related to managing and deploying new software versions. For example, Beattie et al. [2] has looked at the issue of timing the application of security updates, while Crameri et al. [10] has proposed a framework for staged deployment, in which user machines are clustered according to their environment and software updates are tested across clusters using several different strategies. In relation to this work, our approach encourages users to always apply a software update, but it would still benefit from effective update strategies in order to decide what versions to keep when resources are limited.

Many large-scale services, such as Facebook and Flickr use a *continuous deployment* approach, where new versions are continuously released to users [12], [14], but each version is often made accessible only to a fraction of users to prevent complete outage in case of newly introduced errors. While this approach helps minimize the number of users affected by new bugs, certain bugs may manifest themselves only following prolonged operation, after the release has been deployed to the entire user base. We believe our proposed approach is complementary to continuous deployment, and could be effectively combined with it.

VI. Conclusion

Software updates are an integral part of the software development and maintenance process, but unfortunately they present a high risk, as new releases often introduce new bugs and security vulnerabilities.

In this position paper, we have argued for a new way of performing software updates, in which the new version of an application is run in parallel with old application versions, in order to increase the reliability and security of the overall system. We believe that multi-version software updates can have a significant impact on current software engineering

practices, by allowing frequent software updates without sacrificing the stability and security of older versions.

Acknowledgments

We would like to thank Paolo Costa, Peter Pietzuch and Alex Wolf for discussions on multiplicity computing [4], and their feedback on the text. Petr Hosek's doctoral studies are supported by a Google European PhD Fellowship.

References

[1] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, pp. 33–37, 2007.

[2] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright, "Timing the application of security patches for optimal uptime," in *Proceedings of the 16th USENIX Conference on System Administration (LISA'02)*, Nov. 2002.

[3] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'06)*, Jun. 2006.

[4] C. Cadar, P. Pietzuch, and A. L. Wolf, "Multiplicity computing: A vision of software engineering for next-generation computing platform applications," in *Proceedings of the FSE/SDP workshop on the Future of Software Engineering Research (FoSER'10)*, Nov. 2010.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—a technique for cheap recovery," in *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[6] R. Capizzi, A. Long, V. Venkatakrishnan, and A. P. Sistla, "Preventing information leaks through shadow executions," in *Proc. of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, Dec. 2008.

[7] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS'78)*, Jun. 1978.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: end-to-end containment of Internet worms," in *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Oct. 2005.

[9] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proc. of the 15th USENIX Security Symposium (USENIX Security'06)*, Jul.-Aug. 2006.

[10] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in Mirage, an integrated software upgrade testing and distribution system," in *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct. 2007.

[11] D. Devries and F. Piessens, "Noninterference through secure multi-execution," in *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'10)*, May 2010.

[12] R. Harmess, "Flipping out," http://code.flickr.com/blog/2009/12/02/flipping-out/, 2009.

[13] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," Imperial College London, Tech. Rep. DTR11-13, Nov. 2011.

[14] R. Johnson, "OOPSLA keynote: Moving fast at scale - lessons learned at Facebook," in *Proc. of the 24th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'09)*, Oct. 2009.

[15] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, Oct. 2009.

[16] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Oct. 2005.

[17] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

[18] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proc. of the 4th European Conference on Computer Systems (EuroSys'09)*, Mar.-Apr. 2009.

[19] S. Sidiroglou and A. D. Keromytis, "Execution transactions for defending against software failures: Use and evaluation," *Springer International Journal of Information Security (IJIS)*, vol. 5, no. 2, pp. 77–91, 2006.

[20] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Communications of the Association for Computing Machinery (CACM)*, vol. 36, pp. 69–81, 1993.

[21] O. Trachsel and T. R. Gross, "Variant-based competitive parallel execution of sequential programs," in *Proc. of the 7th ACM International Conference on Computing Frontiers (CF'10)*, May 2010.

[22] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Oct. 2011.

[23] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," in *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.

[24] H. Xue, N. Dautenhahn, and S. T. King, "Using replicated execution for a more secure and reliable web browser," in *Proc. of the 19th Network and Distributed System Security Symposium (NDSS'12)*, Feb. 2012.

[25] A. R. Yumerefendi, B. Mickle, and L. P. Cox, "Tightlip: Keeping applications from spilling the beans," in *Proc. of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, Apr. 2007.