A Software Checking Framework Using Distributed Model Checking and Checkpoint/Resume of Virtualized PrOcess Domains

Nageswar Keetha Leon Wu Gail Kaiser Junfeng Yang
Department of Computer Science, Columbia University
{nk2340@, leon@cs., kaiser@cs., junfeng@cs.}columbia.edu

Abstract

Complexity and heterogeneity of the deployed software applications often result in a wide range of dynamic states at runtime. The corner cases of software failure during execution often slip through the traditional software checking. If the software checking infrastructure supports the transparent checkpoint and resume of the live application states, the checking system can preserve and replay the live states in which the software failures occur. We introduce a novel software checking framework that enables application states including program behaviors and execution contexts to be cloned and resumed on a computing cloud. It employs (1) EXPLODE*'s model checking engine for a lightweight and general purpose software checking (2)* ZAP *system for faster, low overhead and transparent checkpoint and resume mechanism through virtualized* PODs *(PrOcess Domains), which is a collection of host-independent processes, and (3) scalable and distributed checking infrastructure based on Distributed* EXPLODE*. Efficient and portable checkpoint/resume and replay mechanism employed in this framework enables scalable software checking in order to improve the reliability of software products. The evaluation we conducted showed its feasibility, efficiency and applicability.*

1 Introduction

Producing reliable software conforming to all specifications for a given software product is a challenging task. Software quality assurance accounts for more than half the cost of software production. Despite increasing reliance on today's computing platforms, large software systems remain buggy, and will continue so in the foreseeable future. Software errors have been reported to take lives [28] and cost billions of dollars annually [34]. Hence, the study of techniques to produce quality software has been active for decades and there have been a wide variety of approaches and tools combining static and dynamic analysis. Techniques based on model checking as a formal analysis of program behavior have been promising for both finding errors and ensuring program properties. Message passing style distributed software checkers [2, 10, 12] speed-up the software checking effort by distributing application states using messaging APIs. Once software failures are detected these tools either capture the execution traces [1] or dump the execution contexts so that developers can inspect and fix the errors. However, software checking systems may not have enough contexts to find the errors as the execution traces don't contain the live context of the execution environment such as file system state, code and data pages.

Several checkpoint/restart mechanisms [16, 17] exist,for example, library based checkpoint-restart mechanisms[18] attempt to replace standard message-passing middleware such as MPI [21, 23]. The library based approaches may be used for a narrow range of applications as they cannot use common operating system services to preserve process level characteristics. However, ZAP and ZAPC [5] provide transparent Operating System level checkpoint-restart mechanisms using kernel-level support and do not require changes to code and allow a checkpoint at any time for the distributed network applications. Capturing partial or full live virtualized states has the following advantages for software checking and error reporting 1) provides fault recovery, 2) enables migration and dynamic load balancing and 3) while checking long running applications, it enables to checkpoint/fork the states for checking and capturing the buggy states.

This framework is implemented by utilizing the following techniques:

Utilizing Distributed EXPLODE's software checking: EXPLODE's engine [1] decides when to fork an execution of a general system to enable execution of different checking paths, and provides techniques for collecting error traces and reproducing errors at developers' sites. EXPLODE allows efficient scheduling of the forked executions, and simplifies creating new checking techniques and combining existing ones into more powerful techniques that find software failures. **Distributed EXPLODE** [2] infrastruc-

ture frees tool builders from re-implementing capabilities of EXPLODE's engine, tool builders focus on the checking logic and may employ a cloud of hosts on demand and check the application states concurrently. EXPLODE takes a lightweight snapshot of the state consisting of state's signature (a hash compaction of an actual state), the trace (the sequence of return values from its path decision function). To restore the state, it replays the sequence of choices from the initial state, however, when the traces are deeper, reconstructing states is a slow, CPU intensive process and in addition, reconstructed states don't constitute their computing environments in which they are incubated. However, utilizing checkpoint/resume technique now allows distributing checkpoints as live OS processes that retain their actual computing environments. It also facilitates the use of distributed hash tables [15, 14] to achieve fair load balancing and avoid redundant checking of the same executions on different machines.

Utilizing the ZAP for Software Checking: Fast, efficient and portable checkpoint/resume and replay mechanisms are needed to capture and replay the live snapshots of application states. In this framework, we implemented these mechanisms through the use of an OS virtualization layer [4] that handles group of OS processes with persistent file system states,network sessions, and device inputs and outputs. By checkpointing the forked executions and resuming them when/where there is an idle resource, we gain flexibility in scheduling. Checkpointing should be faster to clone application states within the normal execution context and ZAP System [6, 5] demonstrates that its checkpoint and resume mechanism is 3 to 55 times faster than OpenVZ and 5 to 1100 times faster than Xen project.

Software checking involves several active states at any point in time, and pulling/migrating of states as POD images on a network file system is expensive, however, in this framework instead of having one EXPLODE process per POD we group several states into a POD so that the migration cost is reduced and distribution of states is faster. The proposed mechanisms will also isolate the side-effects of the executions forked by checking tools from the outside world, and provide application developers the ability to deterministically replay error states since it distributes the application states as live OS processes by forking, cloning and resuming from within the POD. This paper is organized as follows. Section 2 depicts architecture of distributed software checking based on checkpoint/resume of PODs. Section 3 presents implementation and preliminary results. Section 4 discusses related work. Section 5 discusses limitations of current version. Section 6 concludes.

2 Architecture

Software developers and tool builders can leverage this framework by creating test drivers to verify the targeted applications. These drivers specify various choice points via forking using the interface function choose(N) [1, 2]. Once the checking engine attaches to the application at runtime, it creates an initial state of the targeted application and invokes the application in that state. At every choice point where the checked system could perform one of N different actions, it forks *N* child executions to explore each action; each child action is treated as application state and an optional call-back method is provided by tool builders to compute a signature of the forked execution. These signatures form a seen-set (a Distributed Hash Table) and this set is used to discard the executions if their signatures are already in the seen-set. These forking actions make rare executions appear as often as common ones, thereby quickly driving the checked system into corner cases where subtle bugs surface. Checking massive numbers of executions is likely beyond the capability of any single host, however, checking is a search over forked executions, and this search is extremely parallelizable. In addition, EXPLODE performs depth bound search to avoid explosive growth, however, a variety of reduction techniques[27] and heuristics are available to make this search process less expensive.

When an error occurs, system provides a full or partial snapshot of the execution context, so that developers can reliably reproduce the error. Application states can be represented in three variants as 1) a lightweight state consisting of a trace of recorded choices from the initial state, so that system reconstructs the original state by replaying the sequence of recorded choices 2) a data only state consisting of the snapshot of application's data for the system to restore the state by copying the data back 3) a cloned state consisting of the live checkpoint, to restore this state system resumes the checkpoint. Hence in order to cloudify the EXPLODE's software checking infrastructure, if states are represented as traces or data snapshots, then states are distributed via messaging. However, if states are captured as live checkpoints of forked execution contexts then checkpoint/resume mechanism provided by ZAP is used to distribute the cloned states. Moreover, users can adapt this general and unified checking system to enable easy constructions of a stack of new checking schemes and drivers that create opportunities to find more bugs and bugs that cannot be found with each individual checking driver.

2.1 Checkpoint/Resume and Replay Mechanism

Checkpoint/resume and replay mechanisms must support several requirements for software checking: (1) checkpoint and resume must be transparent, i.e., they should not require end-user intervention or application changes; (2) they must handle practical cases such as multiple processes and persistent storage; (3) the checkpointing operation must be fast, because it occurs within the normal execution of a checked system; (4) checkpoint images must be serializ-

able, so our checking framework can migrate them to other machines to balance load, or store them to disk to execute later; and (5) since forked executions are "fake" executions just for checking, they must be sandboxed to avoid any visible side-effects.

In order to meet the above requirements ZAP is an effective tool for checkpoint/resume and replay mechanisms. Fundamental to ZAP's design is the POD abstraction that provides a collection of processes with a host-independent virtualized view of the operating system. PODs have their own private, virtual namespace, and are self-contained units that can be suspended to secondary storage, migrated, and transparently resumed. ZAP checkpoints, resumes PODs without modification or limitations on the use of underlying existing operating system. ZAP does stop the source process for memory migration while pages are copied for checkpointing to destination POD, however, it's a low-overhead mechanism since it occurs in the normal execution context of the source POD. ZAP provides faster checkpointing functionality and PODs decouple the capture of application states from actually checking them; in our framework, we need only make checkpointing fast, and can afford to slow down the migration and resuming of states. ZAP also assumes that commodity operating systems offer loadable kernel modules. However, ZAP doesn't leave any residual dependencies for cloning and migration of POD sessions.

2.2 Distributed Software Checking through PODs

A new initial POD is transparently constructed each time an application is installed or run. When model checking process within a POD generates a new state, a call is made to ZapService, a RPC daemon that runs in user space, from within the model checking process, then ZapService issues commands to ZAP System to clone a new pod by checkpointing and branching from the current state. Since PODs(potentially with multiple processes) can be suspended/resumed, branched and migrated across hosts, PODs are resumed later by available resources. Checking becomes a search over forked executions; this search is extremely parallelizable by adding several hosts. PODs isolate processes from all other applications in user spaces, including other instances of the same application, with its own view of OS services, devices and the file system, preventing conflicts. Figure 1 depicts the overview of the deployment architecture on each host.

Names of the PODs are not unique and to make a POD's name unique, checking framework appends application state's unique identifier to the POD's name when a new state is generated, assigns this POD to a host so that we can resume and run suspended states later by the host. POD contains application-state as well as an instance of EXPLODE's model-checker; however, future implementations may run

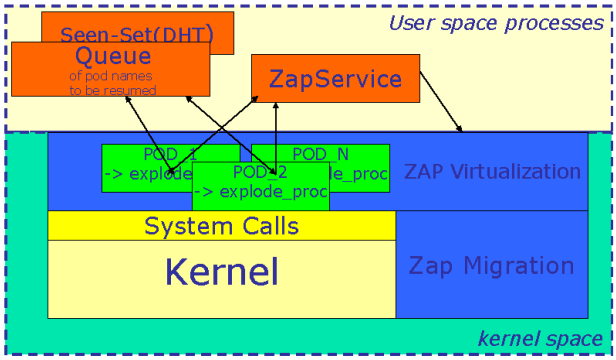

Figure 1. Architecture overview.

model checking engine in user space and actual application states in POD to drive/fork the states while application is being used.

For state distribution this framework, shown in figure 1, uses ZapService, Queuing Service and the Seen-Set: 1) Zap Service is installed on each host and redirects calls to Zap System. EXPLODE processes in the POD issue commands from within the model checking process to ZapService. The cloned PODs can be resumed later by available resources on the same host or can be migrated to distributed hosts, 2) Queuing Service is installed on each host and is a distributed service which runs in user space. Framework drives the state space either in DFS or BFS manner, and assigns each new state randomly to one of the active hosts by sending the POD's reference to a queue on the host for which POD is assigned. An entry in a queue consists of POD's synthesized unique name and its location so that the system can pull the POD and resume it, 3) Seen-Set is maintained in the publicly available OpenDHT storage. Maintaining and updating the visited states (Seen-Set) can slow down the scalability of state exploration because the number of states grows too fast. Hence, for scalability we partition this set based on a DHT so that whenever a host generates a new state it consults the seen-set to avoid duplicate effort. This framework uses OpenDHT's [15] Sun RPC put/get interface over TCP for key and value pairs to be stored, it treats state's signature obtained by MD5 or SHA1 as the key and POD's synthesized unique name as the value.

3 Implementation and Preliminary Results

We have implemented a version of our framework on Linux, we created host machines through VM player to mimic several hosts, we deployed dejaview [7] that implements Zap interface on each VM. In addition, we deployed ZapService, local queuing service (stores the pointer references of cloned and suspended PODs) and used OpenDHT for the Seen-Set (which stores the signatures). We created a sample application driver that attaches to EXPLODE's

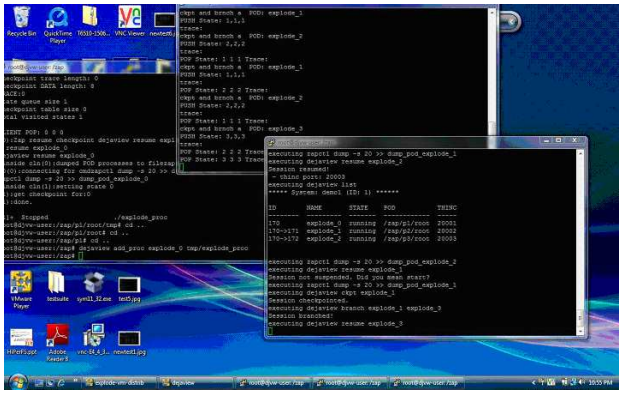

Figure 2. A screen-shot of running services on a single host

model checking engine. To start generating the state space, we created a POD session called eXplode_0 and attached the EXPLODE model checking process(eXplode_proc) which starts in initial state; once this process starts running in the initial POD it triggers the state space generation and exploration, states automatically clone themselves into other PODs upon encountering new states. As shown in the

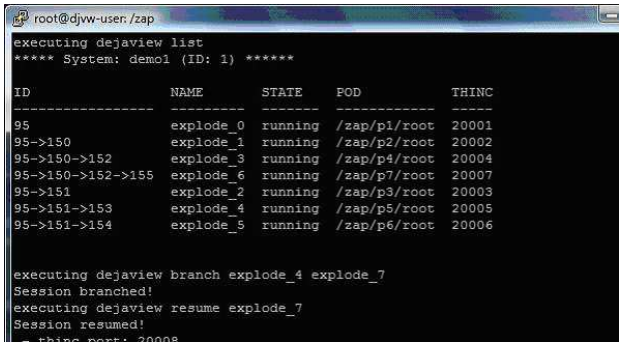

Figure 3. All running pods exploring different states in parallel. Implicitly forms Reachable Graph as a pod tree

left terminal screen in figure 2, "dejaview add_proc eXplode_0 tmp/eXplode_proc" will trigger the state exploration, ZapService daemon shown in the center screen (figure 2) receives commands from within the PODs and executes commands to checkpoint, branch and resume PODs. Once all the states are explored, part of the generated tree of the states is displayed on the right screen in figure 2. eXplode_0 generates two new states eXplode_1 and eXplode_2. eXplode_1 generates eXplode_3, eXplode_2 generates eXplode_4 and eXplode_5, this process continues and we can visually note from the figure 3 that it forms the state search graph consisting of PODs. In figure 3 while PODs are running, using a client utility provided by ZAP system, "thinc_client djvw-user 20002", when remotely logged into the explode_1 POD, figure 4 displays all the processes within the POD. Figure 5 displays the processes in two POD sessions running concurrently and shows running processes of EXPLODE.

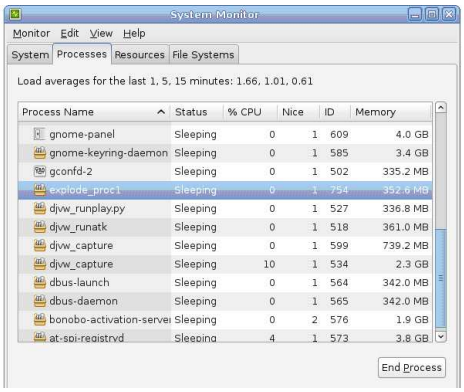

Figure 4. A cloned POD with EXPLODE process inside

Figure 5. Live PODs with EXPLODE processes

4 Related Work

Resource intensive and time-consuming software applications are generally distributed and utilize multiple hosts in a cluster or over a cloud. Software checking tools for formal verification are resource intensive. For Software checking tools, a checkpoint/resume mechanism that is faster and consistent and which runs on commodity operating systems is highly useful. There are several existing approaches [8, 9, 18] for checkpoint, resume functionality. However ZAP ZAPC [5] is more suitable for software checking as it provides low overhead, fast and consistent checkpoint/resume functionality on commodity operating systems. EXPLODE stores states explicitly for checking, checkpoint/re-

sume can be used for distributed checking and error reporting. In general, the stateless model checkers don't explicitly store the states and stateful model checkers store states explicitly. VeriSoft [27] and CHESS [19] are stateless model checkers and consume more CPU cycles. EXPLODE [1], CMC[33], Java Path Finder(JPF), Murphi [12] are explicit state enumerating model checkers and can utilize checkpoint/resume mehanism. The fundamental problem faced by all model checkers is very large number of reachable states (state explosion). Model checkers apply state reduction techniques, apply depth bound on the reachability graph in terms of context switches, path length, number of bugs and exploration time. Murphi, SPIN [10] are also parallelized using message passing style interfaces where the states are distributed as part of messages and reconstructed as approximation to the original state, however, Distributed EXPLODE now supports checkpointing of live states. The in-vivo testing method [32] employs continuously executed tests in the deployment environment. The *Skoll* project [31] has extended the idea of "continuous" round-the-clock testing [37] into the deployment environment by carefully managed facilitation of the execution of tests at distributed installation sites, and then gathering the results back at a central server. Their principal concern is that there are simply too many possible configurations and options to test in the development environment, so tests are run on-site to ensure proper quality assurance. Whereas the *Skoll* work has mostly focused on acceptance testing of compilation build and installation on different target platforms, and thus runs when the application is first deployed at each site, this framework seeks to execute software checking perpetually in the application states including on production operation.

Other approaches include the monitoring, analysis and profiling of deployed software. For example, the *Gamma* system [36] introduces "software tomography" for dividing monitoring tasks and reassembling gathered information; this can then be used for on-site modification of the code to fix defects. The principle difference is that *Gamma* is a monitoring tool that passively measures path or data access coverage, or memory access, and expects users to report bugs. However, this framework checks the applications of interest and automatically report any failures found along with their traces or live checkpoints. Liblit's work on Cooperative Bug Isolation [29] enables large numbers of application instances in the field to analyze themselves with low performance impact, and then report their findings to a central server, where statistical debugging is then used to help developers isolate and fix bugs. Clause [26] has looked at methods of recording, reproducing and minimizing failures to enable and support in-house debugging, and Baah [22] uses machine learning approaches to detect anomalies in deployed software. All of these strategies could take advantage of our framework to enhance their implementation and spread overhead across the clouds.

5 Limitations

In distributing forked executions, security and privacy concerns are more important issues in a wide-area setting, which future versions should improve on it. We plan to explore BambooDHT [15] in our lab for more reliable DHT interface. Error reporting in the framework is not robust as we have not compiled with ZAP API rather we have used dejaview [7] interface for this implementation. The implementation details of synchronization among model checking instances is not fully addressed, this is important since resuming a checkpoint happens at the next instruction after the instruction from where it was branched. Some level of synchronization may be needed to control the execution. Since it's not addressed, there may be some duplicate effort by model checker, and however, we plan to address this in future versions. Checking engine is also made part of POD and cloned along with application state which is not necessary and we plan to decouple this functionality to keep the checking engine out of POD session.

6 Conclusion and Future Work

A number of recent techniques and tools have combined ideas from program analysis and formal methods to make software reliable. We have designed, implemented and evaluated a framework for software checking via PODs on Linux using ZAP and Distributed EXPLODE. This framework utilizes, a thin virtualization layer to decouple applications from the underlying operating system instances, enabling them to checkpoint/migrate across different hosts. It uses scalable and distributed checking infrastructure based on Distributed EXPLODE. We demonstrated the feasibility of checkpointing and resuming live states as part of model checking effort. An application state per POD is expensive while migration due to very large number of sates, however, in our system we can easily group several states or processes per POD and migrate in order to reduce the network overhead. We currently bound depth on the search graph or bound the max number of states/bugs to avoid state explosion. We'll also employ techniques such as partial order reductions to reduce the number of states to be checked. However, this framework transparently and efficiently checks the software using powerful yet easy-to-use unified checking interface, checkpoint and replay checking infrastructure and distributed checking infrastructure. The future evaluation of this system involves building a collection of real checking tools using the framework infrastructure, applying the checking tools on real applications to see their effectiveness, efficiency, performance overhead, and scalability, and comparing these results with other tools.

7 Acknowledgments

We thank Jason Nieh and Oren Laadan for their help on *pod*. Keetha, Wu and Kaiser are members of the Programming Systems Laboratory, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623, EIA-0202063, and NIH 1 U54 CA121852-01A1. Yang is a member of the Reliable Computer Systems Laboratory.

References

[1] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors.

[2] N. Keetha, L. Wu, G. Kaiser, and J. Yang. Distributed explode: A high-performance model checking engine to scale up state-space coverage. Technical Report CUCS-051-08, Columbia University, 2008. In *OSDI '06*. USENIX, 2006.

[3] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *OSDI '02*. USENIX, 2002.

[4] S. Potter and J. Nieh. Reducing downtime due to system maintenance and upgrades. In *19th Large Installation System Administration Conference*, page 4762, 2005.

[5] Oren Laadan, Dan Phung and Jason Nieh Transparent Checkpoint/Restart of Distributed Applications on Commodity Clusters In *PE International Conference on Cluster Computing (Cluster 2005).* Boston, MA, September 2005

[6] Oren Laadan and Jason Nieh Transparent Checkpoint/Restart of Multiple Processes on Commodity Operating Systems In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX 2007).* Santa Clara, CA, June 2007

[7] Oren Laadan, Ricardo A. Baratto, Dan Phung, Shaya Potter and Jason Nieh "DejaView: A Personal Virtual Computer recorder" Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP 2007). Stevenson, WA, October

[8] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C3: A System for Automating Application-Level Checkpointing of MPI Programs In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC03), Oct. 2003*

[9] Y. Chen, J. S. Plank, and K. Li. CLIP - A Checkpointing Tool for Message-Passing Parallel Programs In *Proceedings of the Supercomputing, San Jose, California, Nov. 1997*

[10] G. Holzmann and D. Bosnacki Multi-Core Model Checking with SPIN In *Proceedings of HIPS-TOPMoDRS.* Long Beach, CA, 2007. IEEE.

[11] U. Stern and D. L. Dill Improved Probabilistic Verification by Hash Compaction In *Correct Hardware Design and Verification Methods* volume 987, pages 206-224, Stanford University, USA, 1995. Springer-Verlag

[12] U. Stern and D. L. Dill Parallelizing the Murphi verifier. pages 256-278, 1997

[13] R. Kumar and E. G. Mercer Load balancing parallel explicit state model checking In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification* pages 19-34, Apr. 2005.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications In *the Proceedings of ACM SIGCOMM 2001* San Deigo, CA, August 2001

[15] Sean Rhea, P. Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu OpenDHT: A Public DHT Service and Its Uses In *Proceedings of ACM SIGCOMM'05* Philadelphia, PA, August 2005.

[16] J. C. Sancho, F. Petrini, K. Davis, and R. Gioiosa Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05)* Denver, Colorado, Apr. 2005.

[17] E. Roman A Survey of Checkpoint/Restart Implementations Technical report, Berkeley Lab, 2002. Publication LBNL-54942.

[18] T. Tannenbaum and M. Litzkow The Condor Distributed Processing System. Dr. Dobbs Journal, (227):4048, Feb. 1995.

[19] Madanlal Musuvathi, Shaz Qadeer, Tom Ball, Gerard Basler, P. Arumuga Nainar, Iulian Neamtiu Finding and Reproducing Heisenbugs in Concurrent Programs, OSDI 08.

[20] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2005.

[21] Message Passing Interface [Online]. Available: http://www.mcs.anl.gov/mpich

[22] G. K. Baah, A. Gray, and M. J. Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *Proceedings of the 3rd international workshop on Software quality assurance*. ACM, 2006.

[23] PVM: Parallel Virtual Machine. MIT Press, 2002.

[24] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation*, 2008.

[25] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 319–328, Seattle, WA, USA, 2008. ACM.

[26] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[27] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997.

[28] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[29] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM, 2003.

[30] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *Proc. of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS 2006)*, 2006.

[31] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. Microsoft. `http://support.microsoft.com/kb/283768/`, 2009.

[32] C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach.

In *2nd IEEE International Conference on Software Testing, Verification and Validation*, 2009.

[33] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[34] M. Newman. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[35] U. of California at Berkeley. Open-source software for volunteer computing and grid computing. `http://boinc.berkeley.edu/`, 2009.

[36] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2005.

[37] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 2004.

[38] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[39] Wikipedia. Guan yin. `http://en.wikipedia.org/wiki/Guan_Yin`, 2009.

[40] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. pages 243–257, 2006.

[41] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.