Towards Pre-Deployment Detection of Performance Failures in Cloud Distributed Systems

Riza O. Suminto*, Agung Laksono†, Anang D. Satria†, Thanh Do‡, Haryadi S. Gunawi*

*University of Chicago †Surya University ‡Microsoft Gray Systems Lab

1 Introduction

Modern distributed systems ("cloud systems") have emerged as a dominant backbone for many today's applications. They come in different forms such as scale-out file systems, key-value stores, computing frameworks, synchronization and cluster management services. As these systems collectively become the "cloud operating system", users expect high dependability including performance stability. Unfortunately, the complexity of the software and environment in which they must run has outpaced existing testing and debugging tools. Cloud systems must run at scale with different topologies, execute complex distributed protocols, face load fluctuations and a wide range of hardware faults, and serve users with diverse job characteristics.

One type of important failures is *performance failures*, a situation where a system (*e.g.*, Hadoop) does not deliver the expected performance (*e.g.*, a job takes 10x longer time than usual). Conversation with cloud engineers reflects that performance stability is often more important than performance optimization; when performance failures happen, users are frustrated, systems waste and underutilize resources, and long debugging efforts are required to find and fix the problems. Sadly, performance failures are still common; our previous work shows that 22% of vital issues reported by cloud system developers relate to performance bugs [12].

In this paper, our focus is to answer the following three questions: What is the root-cause anatomy of performance bugs that appear in cloud systems? What is missing within the state of the art of detecting performance bugs? What are new novel directions that can prevent performance failures to happen in the field?

1.1 Anatomy of Performance Bugs

There exists many reports of performance bugs found in deployed distributed systems, but most of them are described in an ad-hoc manner. To dissect root-cause anatomy of performance failures, we perform an in-depth study of performance bugs in Hadoop.

Our finding shows that root causes of performance bugs are *complex deployment scenarios* that the system failed to anticipate. From this, we build a root-cause anatomy (Table 2) that shows some of the *scenario types* (*e.g.*, *DSR*) and *specific conditions* (*e.g.*, DSR_1) that can happen in deployment. For example, with regards to data source selection (*DSR*), some tasks of a job can read from the same datanode (DSR_1) or different datanodes (DSR_2). In terms of data locality (*DLC*), a task can read from a local disk or a remote node. Different hardware faults (*FTY*) such as slow node or network can occur. Hardware faults can happen on different places (*FPL*) such as data, map, and reduce nodes.

Table 2 forms the basis on which we characterize the scenario root causes of performance bugs. That is, a performance bug typically appears in a *specific scenario*. For example, a performance bug surfaces only when an original task and the backup task read from the same slow remote datanode (*scenario:* DSR_1 *&* FTY_1 *&* FPL_1 *&* DLC_1). If one of the conditions is not true, the bug might not surface.

The anatomy and the example above are sample illustrations. The anatomy in Table 2 is far from complete but it is a first step to characterize performance bugs systematically. Later in Section 2, we will present more bug examples and the required scenarios. These examples point to the fact that performance anomalies are hard to find and reproduce. Large-scale cloud systems make many non-deterministic choices (*e.g.*, task placement, data source selection) that depend on deployment conditions. On top of that, external conditions such as hardware faults can happen in different forms and places. The challenge is clear: to unearth performance bugs, we need to exercise the target system against many possible deployment scenarios.

1.2 State of the Art

We ask a simple question: Why do performance bugs keep appearing? Many times similar bugs re-appear (§2). To answer this, we review literature in distributed systems that touch issues related to performance bugs. Table 1 shows the summary of the state of the art.

First, many of existing work focus on *in-deployment* and *post-mortem* tracing, monitoring, debugging, and analysis [2, 4, 9, 10, 22, 23, 26]. Arguably, they rep-

In/Post-Deployment	
Monitoring, Tracing, Profiling	Project 5 [2], Magpie [4], Fay [9], X-Trace [10], LagHunter [16], PiP [22], Spectroscope [23], Log Mining [26]
Pre-Deployment	
Benchmarks	YCSB [5], Limpbench [8]
Model checking	FATE [11], Demeter [13], MacePC [17], SAMC [20], MoDist [27]
Formal methods	P Lang [7], CPN [25], DynamoDB+PlusCal [14, 21]

Table 1: **Categorization of Related Work.**

resent the popular approach but they suffer from one important limitation: *passivity*. In-deployment and post-mortem approaches are passive approaches as they react after performance bugs surface, but they cannot unearth performance bugs prior to deployment.

In terms offline performance testing, one of the standards is running benchmarks [5], which is unfortunately far from simulating real deployment environments. To exercise more scenarios, one can simultaneously run benchmarks and simulate certain environments such as hardware slowdowns in different places, which we did in our previous work [8] and it took hours to observe the result from each experiment (as we must wait to see the impact). In short, performance benchmarking is time consuming and has small coverage.

Regarding to exercising deployment scenarios, there exists many work [13, 27], including ours [11, 15, 20], that permute certain conditions directly on the target system (*i.e.*, "distributed system model checkers"). The downside here is that they primarily focus on reliability but not on performance; they are typically specialized to check classical safety properties. They do not translate well to *time-based* performance verification which requires more time to check; applying the same approaches for performance verification can take weeks to get the result. MacePC [17] is the closest to our work, but it only checks systems written in Mace languages and only permutes timings of concurrent events but not other deployment scenarios such as the ones listed in Table 2.

What we believe missing is fast, pre-deployment detection of performance bugs in distributed systems. One viable approach is the use of formal modeling tools such as Colored Petri Nets (CPN) [25], TLA+/PlusCal [18]. Recently, such an approach is used for verifying cloud systems (*e.g.*, Amazon DynamoDB+PlusCal [14, 21]) but reliability is still the focus (although such tools fit for performance verification). Another downside is that models are "hand-made" in practice; developers must manually model the system logic and scenarios to permute. As an implication, the resulting model may not be a good representation of the real system.

1.3 System Performance Verifier

The journey to highly dependable cloud systems (including performance stability) is ongoing. The use of PlusCal at Amazon hints the need for formal modeling tools to help verify the ever growing complexity of distributed systems. The "hand-made" process is however a major drawback. Therefore, we propose a new advancement: System Performance Verifier (SPV), a framework that takes *real system* code (*e.g.*, Hadoop in Java) and automatically generates the model, environment, and scenarios to permute. The model is based on a modeling tool of choice (*e.g.*, CPN) that has performance verification capability.

To the best of our knowledge, our work is the first that addresses the following question: *How to detect performance bugs in real distributed systems code and do so prior to deployment and in a fast and complete manner?* There are several challenges to address including making the target code amenable for analysis, building a generic system-to-model compiler (Java-to-CPN in our case), optimizing the verification process, and many others (Section 3). Within the last 18 months, we have addressed many of the challenges. A preliminary evaluation of our prototype is given in Section 3.4. In the next section, we first present more examples of complex performance bugs to motivate our vision.

2 Deep Performance Bugs

Using our Cloud Bug Study (CBS) database [12], we further study performance bugs and select the ones that involve buggy logic unearthed in certain deployment scenarios.[1] There are 89 performance bugs (in Hadoop, HBase, and HDFS) that we study carefully, 28 out of which are found in production, while the rest does not have a clear indicator. For brevity, we describe some of the Hadoop performance bugs. We label each bug with a *scenario* (*e.g.*, DSR_1 & DLC_3 & FTY_2 & FPL_1) representing the set of conditions (as shown in Table 2) that must be true to hit the bug. If one of the conditions is not true, the bug might not surface.

• **Untriggered speculative execution.** The heart of tail tolerant systems is speculative execution. When it is not triggered properly, job performance suffers. We find numerous cases where speculative execution is not triggered, resulting in significant job slowdowns. For example, in our previous work [8], we find several flaws in Hadoop speculative execution. The first flaw (*scenario:* DSR_1 & FTY_1 & FPL_1 & DLC_1) is uncovered when an

[1] The bugs covered here were reported between 2009-2013. We study the discussions and patches and ignore "easy" performance bugs (*e.g.*, misuse of Java classes and libraries). The bugs that we cite in this paper contain hyperlinks (*e.g.*, MR-5533).

Scenario Type	Possible Conditions
DLC: Data Locality	(1) Read from remote disk, (2) read from local disk, ...
DSR: Data Source	(1) Some tasks read from same datanode, (2) all tasks read from different datanodes, ...
JCH: Job Characteristic	Map-reduce is (1) many-to-all, (2) all-to-many, (3) large fan-in, (4) large fan-out, ...
JSZ: Job Size	(1) 1 GB jar file, (2) 1 MB jar file, ...
LSZ: Load Size	(1) Thousands of tasks, (2) small number of tasks, ...
FTY: Fault Type	(1) Slow node/NIC, (2) Node disconnect/packet drop, (3) Disk error/out of space, (4) Rack switch, ...
FPL: Fault Placement	Slowdown fault injection at the (1) source datanode, (2) mapper, (3) reducer, ...
FGR: Fault Granularity	(1) Single disk/NIC, (2) single node (deadnode), (3) entire rack (network switch), ...
FTM: Fault Timing	(1) During shuffling, (2) during 95% of task completion, ...
TOP: Topology Scenario	(1) 30 nodes per rack, (2) 3 nodes per rack, ...
TPL: Task Placement	(1) Mappers and reducers are in different nodes, (2) AM and reducers in different nodes, (3) Mappers are in the same node, (4) Most of reducers placed in the same rack, ...

Table 2: **Anatomy of scenario root causes of performance bugs.** *The table lists scenario types and conditions that appeared in the 89 performance bugs that we studied.*

original task and the backup task read from the same (DSR_1) slow (FTY_1 & FPL_1) remote (DLC_1) datanode.

The second flaw (*scenario:* JCH_1 & TPL_1 & FTY_1 & FPL_2) comes up when all reducers must read from a mapper (JCH_1) remotely (TPL_1) and the mapper is slow (FTY_1 & FPL_2); because *all* reducers are slowly pulling data from the slow mapper, there is "no" straggler. But, if the scenario changes (*e.g.*, the job is all-to-many; JCH_2), speculative execution is triggered correctly.

In MR-5533 (*scenario:* FTY_2 & FPL_3 & TPL_2), progress-status heartbeats from disconnected reducers (FTY_2 & FPL_3) do not reach the Application Manager (AM). Here, AM does not trigger backup tasks. In this bug, speculative execution is triggered based on the presence of progress status changes, but not the absence.

The problem of untriggered speculative execution has appeared since the early days of Hadoop (*e.g.*, MR-562).

• $O(n)$ **recovery.** When a *single* failure happens, ideally the recovery should be $O(1)$, but in unexpected situations, buggy recovery logic can be $O(n)$ long. For example, in MR-5251 (*scenario:* FTY_3 & FPL_3 & FTM_1), a reducer receives a disk-out-of-space error (FTY_3 & FPL_3) during the shuffling phase (FTM_1) and reports it to AM which incorrectly treats the exception as a connection problem between the mapper and reducer. Here, AM always "blames" the mapper and runs a new mapper which will communicate with the out-of-space reducer again which then repeats the recovery process n times where n is the configured number of retries.

Another $O(n)$ recovery is in MR-5060 (*scenario:* TPL_1 & TPL_3 & FTY_1 & FPL_2) where a reducer remotely reads (TPL_1) from many mappers (*e.g.*, M1..Mn) that reside in the same node (TPL_3). Here, Hadoop only makes one connection between the reducer and the map node; the reducer will read from each mapper at a time. If the map node is extremely slow (FTY_1 & FPL_2), the reducer only reports to AM about the flaky mapper (*e.g.*, M1) and then continues reading from the next mapper (*e.g.*, M2), ineffectively serializing the recovery of the mappers. Recovery is $O(n)$ where n is the maximum number of mappers that can reside in a node; the number can be large in high-end nodes.

$O(n)$ recovery dated back to early years of Hadoop. For example, in MR-1800 (*scenario:* TPL_1 & TPL_4 & FTY_4 & TOP_1), the mappers and reducers are placed in different racks (TPL_1 & TPL_4) with slow inter-rack switch (FTY_4) which Hadoop does not monitor. Hadoop incorrectly blacklists the map nodes and re-runs the mappers in the same mapper rack (due to data locality). The recovery repeats n times where n is the number nodes in the mapper rack. Interestingly, the problem is not as severe if the number of nodes per rack is small (TOP_2).

• **Long halt from long lock contention.** Sometimes certain operations can be halted unintentionally and must wait for a "big" lock held by other time-consuming operations. For example, in MR-4749, a job operation is holding a lock while cleaning up large temporary data (JSZ_1) while another operation from the same job needs to process a job-commit message. As the commit message is not processed, the job completion is delayed until the background cleaning operation completes.

Similarly, in an earlier bug, MR-1247, a task localization process that downloads big jar files (JSZ_1) holds a lock that prevents the TaskTracker to send heartbeat messages to the JobTracker. The TaskTracker is considered dead, and the corresponding tasks are re-run in another node and hits the same long localization problem repeatedly. Unintentional long lock contention occasionally appears in Hadoop development (*e.g.*, MR-2209, MR-2364, MR-4576, MR-4813).

In summary, performance bugs continue to re-appear with different root causes. There are many more possible scenarios beyond what we list in Table 2. With the anatomy, we manage to describe performance bugs systematically. Our bug descriptions highlight that in order to catch performance bugs prior to deployment, a wide

range of deployment scenarios must be exercised. To do so with speed and good coverage is a major challenge.

3 Towards a New Solution

The previous sections paint the need for a performance verification framework that achieves **four goals**: (1) fast, (2) complete, covering many possible deployment scenarios, (3) runs in pre-deployment, and most importantly (4) directly checks implementation-level code. To the best of our knowledge, there is no framework that achieves all of the four goals.

To further clarify our goals, we are interested in detecting performance failures (*e.g.*, a job takes 5x longer time to finish than expected) along with their root causes, which what we imply as "performance verification". Our focus is not in finding performance sub-optimizations (*e.g.*, opportunities to increase job completion time by 10%).

3.1 Formal Modeling Tools

To achieve the first three goals, one promising way is to adopt formal modeling tools (*e.g.*, CPN, PlusCal). In our work, we chose Colored Petri Nets (CPN) as it is popular in the modeling community and brings significant advantages in performance verification.

First, CPN is generic. One can model almost any system with such tools [6, 24] by simply creating "places", "transitions", and "arcs" containing user-defined functions in Standard ML. Most importantly, CPN incorporates the notion of logical time, allowing us to inject slowdowns, express the expected performance, measure the observed performance, and compare the two.

Second, CPN is fast. It executes the model in logical steps and thus alleviates the cost of setup and cleanup time in testing real distributed systems (*e.g.*, preparing input files, bootstrapping nodes) which can take seconds *per* experiment [8, 11, 20]. Furthermore, Section 2 highlights that many performance bugs surface when there is some hardware slowdown ("limpware" [8]). In direct performance testing, slowdown must be injected in wall-clock time and incurs orders of magnitude longer testing time. With CPN, slowdown can be simulated in logical time and the model "moves forward" rapidly.

Finally, CPN is formal. It has a built-in model checker that can permute all non-deterministic events. We write assertions (*e.g.*, error if a job takes more than 100 steps) and CPN can permute all the defined conditions (Table 2). Note that we do not change deterministic policies in the target code, but whenever some policies use randomness, CPN can permute them.

To make sure this is the right adoption, we manually create CPN models of several protocols (speculative execution, read/write, etc.) that are relevant to surface 18 performance bugs in Hadoop, 2 in HBase, and 2 in HDFS. We then let CPN permute some conditions such as fault placement (*FPL*), data source (*DSR*), and many others. CPN model checker provides the result (*replayable* paths leading to the assertion violations) in just less than 5 minutes. This satisfactory result proves that CPN is powerful enough for our purpose, achieving the first three goals mentioned above. But now, we must face the hardest challenge: achieving the 4th goal.

3.2 Challenges

Although formal modeling tools are powerful, there is *little* adoption within the systems community. Two biggest reasons are that developers must build models manually and the resulting hand-made models do not reflect the real complexity of the systems. Thus, to achieve our 4th goal, we need to build a *system-to-model compiler* that can automatically parse real distributed systems code including their protocols, states, and communications into checkable models. In our case, we need to convert systems written in Java to CPN models. However, these two worlds have different programming paradigms. There are deep challenges both from the programming language as well as the system perspectives.

• *Imperative vs. Functional:* Java is an imperative language while CPN is based on functional language. Developers often write big Java functions as direct changes to stack and heap are easy. Functional language typically requires smaller modular functions. This implies big functions must be re-written into smaller modular functions for direct parsing.

• *Object Oriented vs. Sets:* Systems in Java use objects and a variety of data structures (hash table, list, etc.). CPN represents data only with multisets (sets of key-values where the values can also be sets). For straightforward Java-to-CPN conversion, data representations should be converted into flat data structures

• *By Reference vs. By Value:* Java is all about references, while CPN does not have the same support. In CPN, changing states require writing new key-values to the appropriate set.

• *Complex Dataflow vs. Simple Transition:* One major challenge from the systems side involves complex dataflows and system constructs such as threads, RPCs, heartbeats, queues, locks and condition variables. CPN on the other hand only understands places, transitions, and arcs. Thus, the compiler must convert high-level system constructs into simpler constructs.

• *Wall-clock Time vs. Logical Steps:* Distributed protocols operate on wall-clock time (*e.g.*, timeouts), but CPN works based on logical steps.

Although we specifically discuss Java-to-CPN, we believe the challenges and our solution are applicable to many other system-to-model conversions.

3.3 A Prototype of SPV

We propose System Performance Verifier (SPV), a new framework that takes real system code (*e.g.*, Hadoop in Java) and automatically generates the model, environment, and scenarios to permute. As part of SPV, we have built SysJava-to-CPN compiler in 5305 lines on top of WALA [1]. SysJava implies that the target system must be converted into "SysJava style" as described below. We do not convert arbitrary Java programs to CPN, which is hard to achieve and no such tool exists today. Below, we briefly discuss our high-level methodologies and how we have addressed many of the aforementioned challenges. Due to space constraints, we omit the detailed descriptions of how CPN and SPV work.

• **SysJava**: Our goal is to build a *generic* compiler that can take any Java-based distributed systems without a single change in the compiler. Because of the massive challenges mentioned in the previous section, the compiler cannot simply take vanilla code. Instead, the target system must be re-structured and annotated into a "friendlier" code for the compiler. However, we do not change the program logic. For ease of reference, we name this "SysJava". We have created methodologies to convert Java-based distributed systems into SysJava style, methodologies such as state annotations, flattening object-oriented classes to database key-value styles, code refactoring big functions into CRUD (create-read-update-delete) functions, and many others. This is the main effort that developers need to do to integrate SPV. This process can be potentially simplified if declarative data-centric languages are adopted in the future [3].

• **SysJava-to-CPN compiler:** With SysJava programs ready, the compiler can generate a representative model. Our compiler will parse data flows, function calls, RPC calls, threads, user-defined functions, and all other forms of structures and data communications. For annotated computations and I/Os, the compiler can add logical time (*e.g.*, 1 step). The compiler also marks I/Os that can be delayed and how long (*e.g.*, 20 steps) and treat them as inputs for the model checker.

• **CPN model checker:** Checking the generated model is as simple as clicking a "play button" in CPN. Before that, we easily setup external configurations such as how many nodes to run, how many tasks per job, etc.The compiler already provides to the model checker the scenario types and possible values as shown in Table 2.

Just like any other compiler and verification tools, we note that SPV contains many complex functionalities not fully described in this paper. The complexity is a must as we move the burden from the developers (*e.g.*, manual modeling) into SPV which then results in an overall process that is fast, complete, and automated.

3.4 Preliminary Evaluation

We modified Hadoop MapReduce 1.2.1 in 1067 LOC to convert it into SysJava style. These changes only involve speculation-related components such as job tracker, task tracker, scheduler, and launcher, map and reduce tasks, along with the message communications. We consider the changes minimal as we only re-structure the code but not the logic. Our compiler automatically generates 307 places, 165 transitions, and 733 arcs, collectively 20x larger than our earlier hand-made model. Our SPV can currently permute *TPL* (with 8 conditions), *DSR* (3), *DLC* (2), *DSR* (3), *FTY* (1), and *FPL* (1), with a total of 34 scenarios exercised. We run the CPN model checker with three nodes and one job with two tasks. (In the future, we will scale up the evaluation). The model checker explores 277,847 states and 415,986 arcs and finish with 30 assertion violations that unearth the four performance bugs we inserted. This process initially ran for 1.5 hour but we found interesting optimization opportunities that can significantly reduce the checking time. Overall, SPV is orders of magnitude faster and more complete than performance testing with slowdown injections [8].

4 Conclusion

The complexity of cloud distributed systems and their deployment environments have outpaced existing testing and debugging tools [12, 19]. The use of formal methods has become necessary [14], but the gap between real systems code and hand-made models is still wide. We propose a research direction that bridges the two worlds. We have addressed many important challenges and shown a successful prototype for Hadoop. To show the generality of SPV, we are integrating it into HBase and HDFS. In this work, we focus on performance bugs, but we believe SPV can solve many other deep problems such as distributed deadlock and scheduling problems [12].

5 Acknowledgments

We thank the anonymous reviewers for their tremendous feedback and comments. This material is based upon work supported by the NSF (grant Nos. CCF-1321958, CCF-1336580, and CNS-1350499) and generous supports from NetApp.

References

[1] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[3] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C. Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 2010 EuroSys Conference (EuroSys)*, 2010.

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richar Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[5] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[6] Anna Dedova and Laure Petrucci. From Code to Coloured Petri Nets: Modelling Guide. In *International Workshop on Petri Nets and Software Engineering (PNSE)*, 2012.

[7] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. http://research.microsoft.com/pubs/177118/tr.pdf, 2012.

[8] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[9] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[10] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[12] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[13] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[14] James Hamilton. Challenges in Designing at Scale: Formal Methods in Building Robust Distributed Systems. http://goo.gl/PBF1VK, 2014.

[15] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[16] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch Me If You Can: Performance Bug Detection in the Wild. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[17] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[18] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[19] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. The Case for Drill-Ready Cloud Computing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[20] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[21] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of Formal Methods at Amazon Web Services. http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf, 2014.

[22] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[23] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[24] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] Michael Westergaard. Verifying Parallel Algorithms and Programs Using Coloured Petri Nets, 2012.

[26] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting Large-Scale System Problem Detection by Mining Console Logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[27] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

A Discussion Topics for the Workshop

Below are the discussion points that we hope to bring up at the workshop:

• **Is it time for pre-deployment detection of performance bugs?** For decades, the state of the art of debugging distributed systems performance is by tracing, monitoring and post-mortem analysis. As distributed systems become the backbone of cloud computing, is it time to find revolutionary approaches that find the problems prior to deployment? Is it a big priority for cloud practitioners? Is performance stability as important as availability?

• **Bridging systems code and formal methods.** Are cloud engineers and practitioners willing to adopt formal methods for verifying performance stability? The Amazon DynamoDB team clearly spent deep efforts in adopting PlusCal [14]; it is important because their database is the core of many Amazon services. Are other cloud architects willing to do the same? Is our approach attractive? Is more research in this space needed, where we bridge the big gap between the two worlds of real systems code and formal methods?

• **Future of data-centric languages.** A key to make our approach simple is if the target systems are written in data-centric languages (*e.g.*, [3]). What do cloud practitioners see in terms of the future of data-centric languages?

• **Beyond Hadoop.** Distributed systems are not just about Hadoop. Many people build their own distributed services on top of resources in private and public clouds. In addition, with more "software" in the networking area (*e.g.*, SDN), we believe the concept of finding performance bugs in distributed systems will be widely applicable beyond Big Data systems.

• **Root Cause Anatomy of Performance Bugs.** We believe we have successfully created a structured anatomy of root causes behind performance bugs. Some are depicted in Table 2. We would like to hear more interesting anecdotes from cloud practitioners whether such anatomy is useful.

• **Beyond Performance Bugs.** What kind of other complex distributed system bugs that can be detected by extending SPV?