TOWARDS PRE-DEPLOYMENT DETECTION OF PERFORMANCE FAILURES IN CLOUD DISTRIBUTED SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
RIZA O. SUMINTO

CHICAGO, ILLINOIS
2015
Dedication Text
Epigraph Text
TABLE OF CONTENTS

LIST OF FIGURES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vi
LIST OF TABLES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vii
ACKNOWLEDGMENTS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . viii
ABSTRACT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix
1 INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
2 BACKGROUND . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  2.1 Anatomy of Performance Bugs . . . . . . . . . . . . . . . . . . . . . . . . 2
  2.2 State of the Art . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
3 SYSTEM PERFORMANCE VERIFIER . . . . . . . . . . . . . . . . . . . . . . . 9
  3.1 Formal Modeling Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  3.2 Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  3.3 System Performance Verifier . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  3.4 Colored Petri Net . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  3.5 SysJava Style . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
    3.5.1 Data Flattening . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
    3.5.2 Code Modularization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    3.5.3 Annotation Tagging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.6 Model Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4 IMPLEMENTATION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  4.1 SPV Model Compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  4.2 SysJava Annotation Library . . . . . . . . . . . . . . . . . . . . . . . . . . 21
  4.3 Model Design and Optimization . . . . . . . . . . . . . . . . . . . . . . . . 21
5 EVALUATION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
  5.1 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
  5.2 Preliminary Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
6 RELATED WORKS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
7 CONCLUSION AND FUTURE WORK . . . . . . . . . . . . . . . . . . . . . . . . 31
  7.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
  7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    7.2.1 Subminute Model Checking . . . . . . . . . . . . . . . . . . . . . . . . . 31
    7.2.2 Beyond Performance Bugs . . . . . . . . . . . . . . . . . . . . . . . . . 32
LIST OF FIGURES

3.1 **CPN model of simple task assignment.**  The figure illustrate (a) existence of node and task token in each input places enabling Schedule Task transition, (b) a fired Schedule Task transition, removing token from Node and Task places and adding new token into Task to Run place. ................................................................. 13

5.1 **CPN model of Hadoop MapReduce speculative execution.**  Scaled figure of automatically generated CPN model of Hadoop MapReduce speculative execution protocol. The places in the left side represent system states, while the rest of structure represent code logics or functions. ................................................................. 27
## LIST OF TABLES

2.1 **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. 3

2.2 **Categorization of Related Work.** 6

3.1 **Code modularization and annotation.** The table show changes from original code and its SysJava style. Method containing overlapping logics is modularized into several smaller methods. Each smaller method and system state then tagged using appropriate annotation. 16
ACKNOWLEDGMENTS
ABSTRACT

Today’s cloud systems are susceptible to performance failures, a situation where a system does not deliver the expected performance. Many existing work focus on in-deployment tracing and monitoring and post-mortem analysis, but these methods are "passive" as they only work when performance bugs surface. In our work, we first study many critical performance bugs that had happened in scalable distributed systems such as Hadoop, HDFS, and HBase. We find that performance bugs happen in complex deployment scenarios and we need a fast and relatively complete approach that can verify the performance stability of a target system against many deployment scenarios. We present System Performance Verifier (SPV), which detects performance bugs prior to deployment. SPV takes real system code and automatically generates models that can be fed into formal modeling tools such as Colored Petri Nets. Then, we can permute many possible deployment scenarios and unearth complex performance bugs.
CHAPTER 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 [15].

In this thesis, 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?
CHAPTER 2
BACKGROUND

This chapter gives the motivation of our work. First, we present our study in identifying the anatomy of performance bugs. Second, we present cases of deep performance bugs and motivate the need of performance testing advancement. Finally, we discuss the current state of the art of performance bug detection.

2.1 Anatomy of Performance Bugs

There exist 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 2.1) that shows some of the scenario types (e.g., DSR) and specific conditions (e.g., DSR₁) 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 (DSR₁) or different datanodes (DSR₂). 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 2.1 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: DSR₁ & FTY₁ & FPL₁ & DLC₁). 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 2.1 is
### Table 2.1: 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.

far from complete but it is a first step to characterize performance bugs systematically. 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.

Using our Cloud Bug Study (CBS) database [15], 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

---

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).
Table 2.1) 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 [10], we find several flaws in Hadoop speculative execution. The first flaw *(scenario: $DSR_1 \& FTY_1 \& FPL_1 \& DLC_1$)* is uncovered when an 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., $M_1..M_n$) 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., $M_1$) and then continues reading from the next mapper (e.g., $M_2$), 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 2.1. With the anatomy, we manage
In/Post-Deployment

| Monitoring,    | Project 5 [3], Magpie [5], Fay [11], X-Trace [13], LagHunter [22], PiP [32], Spectroscope [33], Log Mining [40] |
| Tracing,       |                                                                                                           |
| Profiling      |                                                                                                           |

Pre-Deployment

| Benchmarks     | YCSB [7], Limpbench [10] |
| Model checking | FATE [14], Demeter [16], MacePC [23], SAMC [28], MoDist [41] |
| Formal methods | P Lang [9], CPN [39] | DynamoDB+PlusCal [18, 31] |

Table 2.2: Categorization of Related Work.

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.

2.2 State of the Art

We ask a simple question: Why do performance bugs keep appearing? Many times similar bugs reappear (Section 2.1). To answer this, we review literature in distributed systems that touch issues related to performance bugs. Table 2.2 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 [3, 5, 11, 13, 32, 33, 40]. Arguably, they represent 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 of offline performance testing, one of the standards is running benchmarks [7], 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 [10] 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 techniques that exercise deployment scenarios, there exists many works [16, 41], including ours [14, 21, 28], 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 [23] 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 2.1.

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) [39], TLA+/PlusCal [25]. Recently, such an approach is used for verifying cloud systems (e.g., Amazon DynamoDB+PlusCal [18, 31]) 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.

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 3.2). Within the last 18 months, we have addressed many of the challenges. A preliminary evaluation of our prototype is given in Chapter 5. In the next section, we first present more examples of complex performance bugs to motivate our vision.
CHAPTER 3

SYSTEM PERFORMANCE VERIFIER

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 choose 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 [8, 38] 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 [10, 14, 28]. Furthermore, Section 2.1 highlights that many performance bugs surface when there is some hardware slowdown (“limpware” [10]). 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 2.1). 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 (re-playable 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 System Performance Verifier

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. We have built SysJava-to-CPN compiler as main part of SPV framework; 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. We further explain how CPN works in Section 3.4.

• **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 [4]. We will explain this further in Section 3.5.

- **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. We will explain this in Section 4.1.

- **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 2.1. We will explain this in Section 3.6.

Just like any other compiler and verification tools, we note that SPV contains many complex functionalities. 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 Colored Petri Net

Colored Petri Nets (CPN) is a discrete-event modeling language combining the capabilities of Petri nets with the capabilities of a high level programming language [19]. Petri nets provide the foundation of graphical notation and the basic primitives for modeling concurrency, communication,
Figure 3.1: **CPN model of simple task assignment.** The figure illustrate (a) existence of node and task token in each input places enabling Schedule Task transition, (b) a fired Schedule Task transition, removing token from Node and Task places and adding new token into Task to Run place.

and synchronization. It uses CPN ML programming language (based on Standard ML) to provide the primitives for defining data types, describing data manipulation, and creating compact and parameterisable models.

Figure 3.1 shows a CPN model of simple task assignment to node. CPN consist of places, transitions, and arcs. Places are typed and arcs have expressions that may contain variables. In a CPN, places are drawn as ovals and transitions as rectangles. Places and transition can have names; in Figure 3.1, names are drawn inside the figure representing places/transition. Places have types, corresponding to types of variables in standard programming languages; in Figure 3.1, the place Node has enumeration type NODE, while place Task has product type of STRING representing task id and TASKTYPE enumeration representing type of the task. Places additionally can have a marking, a multi-set of tokens (value) residing on the place. The marking of a place before executing transitions is the initial marking.

Places and transitions are connected by arcs having arc expressions. Arc expressions may contain free typed variables. Incoming arc to place/transition is called input arc, while the outgoing arc is called output arc. Given a transition, input and output places each denote the source places of input arcs of the transition and target places of output arcs.

A transition and an assignment of values to all free variables on arcs surrounding the transition is called a binding. A binding is enabled in a marking if the evaluations of all the expressions on
All input arcs result in a multi-set of tokens that is a subset of the token corresponding input place in the marking. A binding can be executed, by removing all tokens from input places according to evaluations of corresponding input arc expression and adding new tokens to all output places according to arc expressions on output arcs. We say that a transition is enabled if there is any enabled bindings of the transition. Figure 3.1a shows an enabled Schedule Task transition. Transitions can additionally have guards (extra expressions written in square brackets next to it). These further limit the transition, as they have to evaluate to true for the transition to be enabled.

In timed CPN Model, we can have timestamp attached in token and colorset definition. Timed token only enabled if the simulation timer has greater or equal value than the timestamp of that token. If there is no enabled token or transition, simulation timer will move forward to the closest token timestamp. Figure 3.1b shows how output token is labeled with timestamp @10. This token will only available when the simulation timer exceed or equal 10 unit of time.

The practical application of CPN modeling and validation relies on the existence of computer tools supporting the construction and manipulation of models. CPN Tools is a well-known program for model editing, simulation, state space analysis, and performance analysis of CPN models. CPN Tools support both untimed and timed hierarchical CPN models. It is available for Windows and Linux and the license can be obtained free of charge via CPN Tools web pages.

CPN Tools support two types of simulation: interactive and automatic. In an interactive simulation, the user is in complete control and determines the individual steps in simulation. CPN Tools shows the effect of executing the selected step in the graphical representation of the CPN model. In automatic simulation, the user specifies the number of steps that are going to be executed, and/or sets a number of stop criteria and breakpoints. The simulator runs automatically and shows only resulting state in the GUI. A simulation report can be saved, containing a specification of the steps that occurred during an automatic simulation.

CPN Tools also provide Access/CPN, a set of library for interacting with the CPN Tools Simulator. Using Access/CPN, one can extend the analysis capabilities or to integrate CPN models into
external applications.

### 3.5 SysJava Style

To bridge this two different worlds of Java and CPN, what we want is a system-to-model compiler based program analysis. But just program analysis is not enough. We need to apply some code restructuring and design pattern so the compiler can work. This includes data structure flattening, code modularization, and annotation tagging. The restructured and “friendlier” Java code from this process is what we call as **SysJava**.

#### 3.5.1 Data Flattening

CPN model is both state and action oriented. It describes the system states and events that cause system to change state. In CPN, states are represented as tokens, colorsets, and places, while events or actions are represented as bindings, transitions and arches. Thus the first thing we need to do is to make a *clear separation of the states and actions in system code*.

In Java code, most of system states are represented in data structures objects such as Map, Tree, and so on. However CPN does not have the same notion. CPN represents data as simple multisets and requires transitions to actually fetch the data for every update. Complex petri net structures are required to fully represent advance data structures, which is a non-trivial task for SPV compiler to generate.

To address this in SysJava, we need to modify the systems data structure into tables of values and parse them into simple CPN List colorset. If an object has multiple hierarchical values, we flatten it into multiple relational tables, just like databases. For example, In Hadoop code, we may have a list of Job, where each Job object has a list of Task objects inside it. In SysJava, we will need to write that as three list: Job list, Task list, and list of mapping between Job and Task. Each of this flattened data structure will be parsed as different places in CPN. An action that needs access to system states such as state update will need to fetch and set value to this places.
Table 3.1: **Code modularization and annotation.** The table show changes from original code and its SysJava style. Method containing overlapping logics is modularized into several smaller methods. Each smaller method and system state then tagged using appropriate annotation.

This might sounds like a radical change on the target system. But this simplification is strictly required due to the limitation of Java and CPN itself. We believe that such change is bearable, as it does not change the logic of the system itself. We also believe that this data flattening approach will be much friendlier if applied against data centric or functional languages in the future.

### 3.5.2 Code Modularization

The next thing SPV compiler needs to do is to transform Java functions into CPN transitions. However, there are many big functions containing multiple interleaving logic in system code.

Big functions are broken down into several small-modular functions. We refactor the code such that control flow logic like “Loop” or “If-Else-Branch” is contained in single separate function. As
we flatten system data structure in the previous step, we also convert methods accessing system
states into simple Create, Read, Update, Delete (CRUD) methods. We call this method as state
modifier method.

CPN is based on functional language. Therefore, in this refactoring phase, we need to write the
Java code such that it is close to functional style. Each method is written as a function that accepts
a set of input parameters and returns a value. We avoid accessing global variables or fields from a
method except for state modifier method. Thus, all access to global variables need to go through
the state modifier methods.

Later, each of these functions and data structures will be parsed into a submodel, which then
will be connected together as a building blocks to create a complete system model.

3.5.3 Annotation Tagging

Modeling the whole system as one big model is impractical, both in parsing and model checking.
Sometimes, only a subset of the system is needed to verify the protocol that we want to check. In
order to guide SPV compiler to focus on this important subset, we mark the code with Java anno-
tation. The annotation is also used to provide additional knowledge about the code to compiler.

This annotations fall into four category:

- **Data structure annotation.** This is to tag a field or class that represents system data struc-
tures for colorset declaration and system state parsing. Most of Java primitives type such as
int, double, or string, have a counterpart type in CPN primitive colorset. For other compos-
ite types or classes, SPV compiler need to analyze subclasses that construct them and then
declare a new CPN colorset based on available CPN primitive colorset. In SysJava, @Data
annotation shall be used to tag this type of class.

Table 3.1 shows another data structure annotation, @State. A class field tagged with @State
annotation will be parsed into a CPN place representing a system state. All of the system
state places collectively represent a model state that will be verified later in the model checking process.

- **I/O annotation.** Annotation in this category is used to hint an I/O process such as disk read or RPC call. Table 3.1 shows RPC annotation used to tag heartbeat method, which is an RPC method between TaskTracker and JobTracker. I/O annotation also has a special feature to define latency, slowdown or failure that might happen in an I/O.

- **CRUD and process annotation.** This is mainly used to capture transition logic, annotation in this category is used to tag method where state modification and processing occur. Control flow logic and state modifier methods that do either Create, Read, Update, or Delete to system state shall be tagged according to its type. Table 3.1 shows @ForEach annotation for iteration logic, while @Update annotation assigned to method that update system state tasks.

- **Miscellaneous annotation.** The rest of annotations that are not related to system logic fall into this category. Miscellaneous annotations mostly used as an additional hint to overcome some limitation in underlying static analysis library and help guide function call exploration, such as limiting or broaden static analysis scope.

### 3.6 Model Checking

The output model generated by SPV compiler is executable in CPN Tools. Prior to model checking, we need to define external configuration such as workload to run, number of nodes, topology, and so on. Defining this configuration is as easy as writing initial value of places in CPN Tools interface.

There are two ways to check a model in CPN Tools. The first is by simulation, which is similar as doing a single run of the system. Users can click play or next button in CPN Tools interface to move the simulation forward. Users also can click enabled transition, marked with green shade,
individually to choose which transition should go first when there are multiple transitions enabled at the same time. Simulation is good for quick and interactive checking, but not able to excercise multiple scenarios explained at Table 2.1.

The second way to check model in CPN Tools is by running state space analysis. Different from simulation, state space analysis will explore all possible executions from an initial state of CPN Model. It works by capturing current values of all places as global states of model. We call each state as marking. A successor marking is called reachable from predecessor marking if there exist a binding or transition that can change state of predecessor marking to successor marking. A marking having no enabled transition is said to be dead marking. State space is a directed graph where the nodes correspond to set of reachable markings and arcs correspond to occurring binding element (transition).

This binding in CPN model is equivalent to non-deterministic events of system, such as task to node mapping in scheduling. For example, if we have two available nodes A and B, and one task in Hadoop MapReduce, the task might be assigned to either node A or B, which will lead to different system states. Similarly in CPN model, a marking may have multiple successor markings which may happen from executing different enabled bindings. Thus, state space analysis can exercise multiple scenarios condition listed in Table 2.1.

From this state space, we can verify the performance of model by checking the marking (usually the dead marking) against a pre-defined specification, such as “a model is acceptable if it can run the job no more than 100 steps” or “a system performance is acceptable if it can run the job no more than 3X unit of time” where X is expected system performance. We say that a state is caught in performance failure if it has specification violation. From the violating state, we can further backtrack the buggy path to find the root cause that lead to performance failure.
CHAPTER 4
IMPLEMENTATION

In this chapter, we describe our SysJava-to-CPN compiler prototype, which we built in 5305 lines of Java code, written on top of WALA [2] library for static analysis and AccessCPN [1], provided by CPN Tools, for CPN model generation. We also implement SysJava annotation library for programmers in order to help code refactoring to SysJava style that understood by SPV compiler. We will then explain in detail about some design choice and optimization that we implement in SPV compiler.

4.1 SPV Model Compiler

Given the SysJava version of target system code, SPV compiler will work in two phases. First is analysis phase, where SPV compiler will do couple of static analysis over SysJava code to extract the system construct and logic. Second is the modeling phase, where it will start drawing CPN places and transitions and connecting each of them based on information extracted on analysis phase.

In analysis phase, the compiler will first analyze the class hierarchy to capture type and system states. For every class tagged with data structure annotation, it will create new CPN colorset definitions. It will memorize detected system states and later parse it to a CPN place with appropriate colorset.

Next thing the compiler will do is to explore data flows and function calls of the system. A function call graph of system will be generated. A node in this graph represents an annotated function in system code, while edge between two nodes represent function call.

In the modeling phase, the compiler will extract the logic of each node in function call graph and map each of it into CPN structure. We do that by matching the annotations to template patterns we have, similar like model extraction from previous works [36, 39]. We implemented 16 patterns
for different functionalities such as control flow logic like loop and branch, CRUD for system states, process creations, computation logic, and so on. We call CPN structure generated from this step as submodel. Each submodel will have set of input and output places and may also include another submodel inside it. Finally, the compiler will build complete system model by connecting submodels to each other, based on function call graph. An output from SPV compiler is a CPN model in XML file.

4.2 SysJava Annotation Library

In order to assist code refactoring to SysJava style, we also implement SysJava annotation library. Note that this library is not special class library for SPV, but just a set of Java annotation. In the annotation step of SysJava refactoring, programmers will need to tag the modularized code with annotation from this library. We have implemented 24 annotations that will help compiler map tagged function to one of 16 template patterns.

Some of annotations have extra field where programmers can define additional information for model compilation. For example, in I/O annotation, programmers may define how long the I/O will take, and the compiler will adjust the time inscription for that function transition according to supplied value. Programmers are able to define custom function or initial value of system states via annotation. In this case, programmers will need to write it as SML declaration in the annotation field, and compiler will insert it to generated model.

4.3 Model Design and Optimization

There are two problems that we face during modeling. The first one is about scalability of the model such as how we model the system if we have multiple instances of worker nodes. The second one is how to reduce state space explosion during state space analysis.

To solve the first problem, there are two ways to design the model: modeling component as
structure or component as token. Modeling component as structure means we will model each instance of system component as independent petri net structure. For example, if we want to model Hadoop cluster with 10 workers, we should create a petri net model for one node and copy it 10 times. This is the easiest way to model system, but resulting model will not scale well. Generated model will contain fixed number of component structure and we can not copy or remove petri net structure during simulation or model checking. On the other hand, when modeling component as token, we only require one petri net structure of the component and represent multiple component as token that run inside the structure. The downside of this approach is we will require each component to have unique identity and we need to keep track this identity for every transition so that a transition of one instance will not steal token that belongs to different instance. Since SPV framework relies on compiler for model generation, we choose to model component as token, as we can leave the burden of token identity tracking to compiler.

Like many other state based model checking, we also face state space explosion problem, a condition where the state space grow very large. Most of timed petri net, including generated model by SPV compiler, will have a directed acyclic graph. There will be a lot of branches from one state, i.e. due to concurrency between multiple instances, but many will converge into a common final state.

In case where concurrency is not affecting the verification result, we can employ two optimizations: delayed transition and prime number interval. In delayed transition, we increment the timestamp of every output token by small delay, typically 1 unit of time. When there are multiple transition enabled at one time, the output token from first fired transition will not be available until the rest of enabled transitions are fired too and model clock proceed.

Multiple transitions are enabled because of many instances, i.e. mapper or task reporter thread, alive at the same time. Prime number interval is an optimization that add small delay to token inactivity duration, like thread sleep or read latency, so it will sum to closest prime number, thus reducing the chance of two token alive at the same time. For example, if a heartbeat report supposed
to be send every 1 second, we model the thread sleep time as 1009 unit of time. If map iteration takes 1.5 second, we model the latency as 1511 unit of time, and so on.

These optimizations can effectively reduce branches in state spaces, which lead to reduced number of states and arches to explore, in cost of reducing concurrency level. If full concurrency is expected in model checking, this optimization can be disabled or applied only on specific part of model.
CHAPTER 5
EVALUATION

We now evaluate SPV by presenting experimental result that answer following questions:

1. Can SPV catch deep bugs?

2. How fast is SPV in finding deep bugs?

To answer those questions, we evaluate SPV framework against speculative execution protocol of Hadoop MapReduce. 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, task launcher, map tasks, and the message communications. We consider the changes minimal as we only re-structure the code but not the logic.

5.1 Experiment Setup

Our compiler automatically generates 361 places, 208 transitions, and 905 arcs, collectively 20x larger than our earlier hand-made model. Figure 5.1 shows the scaled view of the generated model.

In this experiment, we want to see if speculative execution able to maintain performance when a failure introduced to system. We set a single bad data node in model so that if a mapper task read from this bad node, its progress will be 10 times slower than reading from healthy data node. A mapper modeled to do 10 iterations of read with 10% progress increase per iteration. We set latency and timing as 1.5 second for one map iteration, 1 second for heartbeat interval, 1 second for task launcher thread sleep, and 3 seconds for map-to-tasktracker progress reporting, where 1 second equivalent as 1000 units of time in model time.

We run the CPN model checker with two nodes and one job with two tasks (in the future, we will scale up the evaluation). Under failure-free deployment scenario, we can roughly estimate a job to finish after 15,000 units of time. And since we introduce single bad node to test speculative
execution in this setup, we increase acceptable execution time two times of normal execution, and
defined a specification as "model performance in acceptable if a job can complete in no more than
30,000 units of time“, and a stopping criteria to halt state space exploration when model time reach
30,000 unit of time. The specification is written as SML function and evaluated against all dead
marking states. Job is said to be complete if both job average progress and all its tasks progress
reach 100%. This model will permute TPL (with 8 conditions), DSR (2), DLC (2), FTY (1), and
FPL (1), with a total of 16 scenarios exercised. The experiment was run on CPN Tools on top of
standard MacBook machine.

5.2 Preliminary Result

The model checker explores 125,977 states and 183,639 arcs. The state space exploration fin-
ished with 12 dead markings, where number of steps from initial marking to dead marking varied
between 6,477 to 7,717 steps. We capture the performance bugs by inspecting the system state
containing the job progress and tasks progress on the 12 dead markings and check them against the
specification. 6 of the dead markings pass the specification, finished with 100% job progress. The
other 6 got specification violation, where 4 dead markings stopped with 55% job progress, and 2
dead markings stopped with only 10% job progress.

To further check the root cause of bugs, we trace back the violating state up to the ancestor
state that also has branch to acceptable state. We then compare the difference between branches
that lead to violating state and branch that lead to acceptable state. From this analysis, we found
two root causes that lead to performance bug:

• Both task hit the bad data node, thus both progressing in the same slow pace and never
trigger speculative execution (2 dead marking with 10% job progress).

• Speculative task hit the same bad data node. At first, one of the task hit the bad data
source. Faster task can trigger speculative task, but the speculative task hit the same bad data
node like original slow task (4 dead marking with 55% job progress).

The model checking process runs for about 30 minutes. We still consider this slow as the size of workload is quite small. However, we found another 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 [10].
Figure 5.1: **CPN model of Hadoop MapReduce speculative execution.** Scaled figure of automatically generated CPN model of Hadoop MapReduce speculative execution protocol. The places in the left side represent system states, while the rest of structure represent code logics or functions.
CHAPTER 6

RELATED WORKS

Performance bug is a software defects where relatively simple source-code changes can significantly speed up software, while preserving functionality [20]. In this work, we focus on performance bugs that occur in cloud distributed system, which caused by complex deployment scenarios that the system failed to anticipate.

There are many works in performance monitoring, tracing, and post-mortem debugging. Aguiler et al., [3] find performance anomaly in distributed system by treating each component as black box, and inferring the operation paths by looking at message traces. Magpie [5] is a toolchain that works with events generated by operating system, middleware, and application instrumentations, cluster similar behavior, and inferring causal relations from total ordering of events. Pip [32], also use causal path, but instead of relying on statistic and inference, it uses explicit path identifiers and programmer written expectation to gather and compare actual and expected behavior of distributed system. X-Trace [13] is another tracing framework that works by inserting trace metadata into application task request and propagating it down through modified protocol interfaces, allowing it to reconstruct user’s task tree. Spectroscope [33] capture anomaly by tracing request flow (path and timing) of execution across the components of a distributed system, and compare it with one execution that serve as a model of acceptable performance. Fay [11] employs dynamic instrumentations that insert function tracing into to user-level process or the operating system kernel, aggregates measurement, and provides analysis mechanisms specific to user query, written in a declarative language FayLINQ. Xu et al., [40] focus more on mining existing log. They propose a framework that analyze source code to extract semi-structured data automatically from legacy text logs and applied anomaly detection, based on Principal Component Analysis, on features extracted from logs. These in-deployment and post-mortem approaches, arguably, still the most popular approach to capture performance bugs in distributed system. However, they only react when performance bugs surface and cannot unearth performance bugs prior to deployment.
Benchmarking is another standard in performance testing. Yahoo! Cloud Serving Benchmark [7] is a popular benchmarking framework, initially created to test PNUTS system of Yahoo!. The framework consists of a workload generating client and a package of standard workloads that cover interesting parts of the performance space. To exercise more deployment scenario, one can simultaneously run benchmarks and simulate certain environment events such as hardware slowdowns. Limpbench [10] follow this strategy by running microbenchmarks and inject slow NIC/disk to test several protocol in five cloud system. However, it took hours to observe the result fro each experiment. In general, performance testing is costly, time consuming, and has small coverage.

Programmers may find bugs using model checking. Model checking is exhaustive, covering all possible behaviors of system execution. MoDist [41] is arguably one of the most powerful distributed system model checkers. It employ dynamic partial order reduction [12] which exploits the independence of events to reduce the state space explosion. Demeter [16] improve MoDist by introducing dynamic interface reduction (DIR). DIR records local exploration and replays future incoming messages without the need for global exploration. SAMC [28] is a white-box model checking approach that takes semantic knowledge of how events are processed by target system and incorporates that information policies. Leveraging semantic knowledge allows SAMC to employ four semantic-aware reduction policy: local-message independence, crash-message independence, crash recovery symmetry, and reboot synchronization symmetry. Besides model checkers, there exist sophisticated testing frameworks for distributed system that able to permute multiple failures such as FATE [14], and PreFail [21]. So far, these model checking and testing framework has primarily focus on reliability, typically specialized to check classical safety properties, but not on performance. They do not translate well to time-based performance verification, which requires more time to check. Mace [23] is closely related to our work. They do performance model checking and check implementation directly. However, the checked system needs to be written in Mace framework and language. They permute timing of concurrent events, obtained through training, but not other deployment scenario such as the one listed in Table 2.1.
Formal modeling tools such as Colored Petri Nets (CPN) [39] or TLA+/PlusCal [25] has been employed to do fast verification in distributed performance. Amazon successfully used TLA+/PlusCal to verify reliability of DynamoDB [18, 31]). Programmers need to write system specifications manually in TLA+ or PlusCal language, and check the specifications using TLC model checker. Similar like model checkers reliability is still the focus, although such tools also fit for performance verification [26].
CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

The complexity of cloud distributed systems and their deployment environments have outpaced existing testing and debugging tools [15, 27]. The use of formal methods has become necessary [18], 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 [15].

7.2 Future Work

There are many works and improvement that should be addressed to further close the gap between real system implementation and formal model. In this work, we have built an SPV compiler that enable automatic parsing from restructured Java code into formal model, along with methodologies required for code restructuring. Next phase in SPV development is how to create more optimize model that can lead to even faster model checking? How to build model that able to scale up well to workload increase? We will address the question in this section. In addition, we will discuss the possibility of implementing SPV outside of performance verification.

7.2.1 Subminute Model Checking

So far, our work has been focused on how to correctly model real system into CPN Model. Our prototype for Hadoop show that SPV can achieve that target, generating larger and more detail model than our previous hand made model.
However larger model size also means longer model checking time. Every transition that fired in model checking will generate new state. Therefore, the more transitions exist in model, the more states will be produced by model checker. We believe that smaller number of states will lead to faster model checking. We have implement a compiler optimization that carefully add limit to token concurrency in generated model, but this only effective in decreasing the diameter size of state spaces (the branches), not the depth.

There are numerous techniques on time petri net reduction that can be embedded SPV compiler [17, 30, 34, 37]. In our preliminary result, we found that many CPN structures in generated model can actually be compacted into more concise structure. For example, the heartbeating process that dominate our Hadoop MapReduce model is actually a long synchronized chain of submodels that never execute multiple concurrent tokens. One optimization that can be done in compiler is by compressing this submodel chain into single submodel containing fewer transitions. Logic functions from merged transitions can be written as single SML function. Thus, a routine that should take multiple transitions can be expressed in single transition, generate only one new state instead of multiple intermediary state.

Another solution is to employ parallel model checker [6], which will require us to modify the CPN model checker further.

\[ \text{7.2.2 Beyond Performance Bugs} \]

SPV is focused on capturing performance bugs. Therefore our works has largely focus on analyzing time property of the model.

As SPV is built on top of CPN, it also carries all good features that exist in formal method tool. It able to pulls other behavioral properties of model such as reachability, boundness, liveness, and fairness properties. Given this ability, we believe the methodologies that we use in SPV also able to help programmers solve deep problems in other areas such as distributed deadlock, scheduling, and resource allocation. Instead of time, we can adjust the specification to analyze the behavioral
properties for these problems.

Petri nets itself has been widely used in many distributed system area, including reliability, concurrency validation, and deadlock avoidance [19, 24, 29, 35].
REFERENCES


