Anomaly Detection and Fault Localization Using Runtime State Models

by

Xi Cheng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software systems are impacting every aspect of our daily lives, making software failures expensive, even life endangering. Despite rigorous testing, software bugs inevitably exist, especially in complex systems. Existing tools to aid debugging, such as tracing, profiling, and logging facilities, reveal the behavior of a program's execution; however, they require the developers to manually correlate the data to diagnose faults.

This work is the first to introduce the *Runtime State Model*, a summarization of a program's behavior, for software anomaly detection and fault localization. A *Runtime State Model* is constructed from variables' value change events of an execution. It consists of a set of states, and state transitions, where a state is a set of variables with their current values, and a state transition is induced by a variable's value change. Comparisons between states from difference executions can be conducted to detect software anomalies. Deviations from the healthy states also help explain and locate faults in the source code. To automate this process, we implement *Xtract*, a facility that automatically extracts runtime traces from the Java Virtual Machines and constructs *Runtime State Model*s for multiple simultaneous Java applications. Our evaluation provides evidence that *Runtime State Model*s might be effective in detecting and locating injected faults to a RUBiS server with *Xtract*.

Acknowledgements

I would like to take this opportunity to express my appreciation to my supervisor Paul A. S. Ward for introducing me to the world of research and for his guidance and support throughout my Master's degree program. I am grateful to Bernard Wong and Lin Tan for being my thesis readers, whose constructive comments and feedback helped improve the work.

This thesis is made possible with valuable assistance from many individuals since the beginning of the project. I thank the members of the Shoshin Lab for sharing their thoughts and providing insightful discussions. My appreciation also goes to the staff of the CSCF and the Department of ECE for their technical and administrative assistance.

To my lovely friends and those who had been part of my life, thanks for your inspirations, enthusiasm, friendship and love. This journey would not have been as enjoyable without all the fun and laughter we had, through ups and downs.

My deepest gratitude goes to my mom, Xiaozhen Wang, for her selfless love, unconditional support and encouragement along the way.

Dedication

In memory of my dad, who is often in my thoughts.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivations

With the popularity of computers, mobile devices and easier access to the Internet, software systems are impacting every aspect of our daily lives, making software failures expensive, even life endangering. Despite rigorous testing in almost all production software, software bugs still inevitably exist, especially in the complex systems. Take Boeing's newest 787 Dreamliner as an example: it was discovered, after nearly 4 years in service, that an integer overflow bug in its generator control unit would cause a complete electric shutdown and potentially loss of control of the aircraft [29].

For enterprise entities, software failures result in capital losses. In August 2012, Knight Capital Group lost $440 million in 45 minutes due to a software failure, causing its stock price to drop by nearly 60% in one day [54].

A survey conducted from October to November 2014 reveals that [22],

- the average cost of unplanned application downtime per year is $1.25 billion to $2.5 billion for Fortune 1000 companies.
- the average hourly cost of an infrastructure failure is $100,000.
- the average cost of critical application failures per hour is $500,000 to $1 million.

Unfortunately, implementing reliable software has never been easy, even for the giants. On September 20, 2015, Amazon's web services (AWS) experienced a 5-hour long outage, causing service interruptions for many of its big customers, including Netflix, Airbnb, IMDb, and Amazon's own online markets [17], despite of its commitment to provide a minimum of 99.95% monthly uptime in the Service Level Agreement [32].

Approaches to improve software dependability has been extensively explored, by both Systems and Software Engineering communities. As a couple of representatives, replication techniques create and coordinate replicas to increase system availability [1, 12, 37], rollback recoveries restore the failing processes to a state before the presence of failures [13, 34, 35], crash only software is designed that its components could be restarted without prior synchronizations [8, 9, 45], software rejuvenation prevents the occurrence of failures by proactively cleaning up the system's internal states [26, 31]. These approaches are stepping stones towards today's dependable services. They reduce the end-to-end visibility of failures, improving the MTTF (Mean Time To Failure) by orders of magnitude, however, will not help in fault diagnosis, i.e., debugging.

Runtime tracing is the most intuitive approach when it comes to debugging. The basic form of runtime tracing that most are familiar with is to use a debugger, e.g., *gdb*, to step over each line of the source code while inspecting variable values to find the bug. In this case, debugging is a process of inspecting a program's execution paths and variable changes, i.e., states, that deviate from a developer's expectations.

Efforts to ease this process focus on providing mechanisms to reveal the inner workings of a program through static instrumentations [30], dynamic instrumentations [11], or OS events tracking [2, 7]. In the context of a networked system, approaches also consider request flows between components [24, 51]. Runtime tracing is often perceived as a mature technique extensively used even in production systems [11, 51], however, these approaches merely provide means to retrieve runtime data and expect the developers to manually correlate the data to diagnose faults.

The automation of diagnosing program faults and anomalies is usually achieved through behavior matching, where the behavior of a program is defined as observable effects in its execution [5]. Early work towards automated fault detection takes descriptions of programs' expected behaviors as input [5, 47, 49], and therefore requires great efforts from the developers to manually define their expectations.

This limitation is addressed by the automated construction of program behavioral models. It is observed that program behavioral models presented in existing work fall generally into two categories,

- In the context of performance diagnosis, models are constructed from performance costs, including, system resource consumptions [4], the running time of system calls [3], time spent on network requests [50] or information recorded in the log messages [44, 60], e.g., timing, number of records. However, these models can not be applied to the diagnosis of non-performance related issues.

- To detect more generic types of faults, flows and paths are usually used to model a system's behavior. For the case of diagnosing componentized systems, request flows and status are adopted in anomaly detection [10, 14], but they are implementation-agnostic, and can only be used to determine the failing components. Attempts to model execution paths suffer from high false positives [27].

To address these limitations and achieve the automated detection of generic faults, at a source code granularity, we propose the use of *Runtime State Model*s in anomaly detection and fault localization. To define a *Runtime State Model*, we first refer to Lamport's definition of a *computation*, that a *computation* is a sequence of *step*s that result in transitions of *states* [36].

In the context of software, the following observations are made,

- a *state* is represented by a set of (*variable, value*) pairs,
- a *step*, i.e., a transition of *state*s, is induced by the change of a *variable's value*,
- and therefore, a sequence of *variable's value* changes defines a program's *behavior*.

This leads to one of the fundamental arguments of the work, that

Argument: Given that a runtime trace, i.e., *sequence* of variable's value changes, *defines* the program's behavior, a model constructed from a runtime trace to include a *set* of its states, and a *set* of state transitions, *summarizes* the behavior of an execution. We define the model as a *Runtime State Model*.

Note that a runtime trace captures the temporal order between the changes; however, the *set* of transitions in a *Runtime State Model* ignores their temporal property. It is also worth noting that to preserve the compactness of a *Runtime State Model*, each state in the model could be a subset of the program's corresponding states to exclude outliers. To summarize the common behavior of a program, a model can be constructed from multiple runtime traces of the same program to include the shared states and transitions.

Intuitively, a *Runtime State Model* gradually constructed from a healthy execution, or a collection of healthy executions (of the same program), consists of *healthy state*s that summarize the program's healthy behavior. By comparing the states of a failing execution to the healthy states, we could derive a set of anomalous states, i.e., states that deviate from the healthy states, and a set of transitions that eventually lead to the anomalous

states. Since each state transition is a variable's value change, it helps us locate the fault back to the source code.

The advantages of using *Runtime State Model*s in anomaly detection and fault localization include,

- Anomalies detected with a *Runtime State Model* could be mapped back to the source code, achieving fault localization at a source code granularity.
- As opposed to models constructed from particular metrics, a *Runtime State Model* is constructed directly from runtime traces, which makes it capable of detecting generic faults.
- Constructing a *Runtime State Model* is an application-agnostic process and does not require knowledge of the system structure or source code beforehand, which also makes this technique applicable to all types of applications.

To the best of our knowledge, we are the first to formulate the notion of *Runtime State Model*s in the context of anomaly detection and fault localization. This work focuses on the automated construction of *Runtime State Model*s, and showing evidence that *Runtime State Model*s might be effective in detecting runtime anomalies.

We recognize the following novel and significant contributions,

1.1 Contributions

- This work is the first to formulate the notion of using *Runtime State Model*s in the context of software anomaly detection and fault localization.
- We present *Xtract*, a facility that automatically extracts runtime traces from the Java Virtual Machines and constructs *Runtime State Model*s for multiple simultaneous Java applications. The facility includes,
  - A *Runtime Data Extraction Infrastructure* that retrieves runtime traces directly from the Java Virtual Machines through a set of JVMTI constructs. As an effort to extract local variable change events from the JVMs, we implement the local variable watchpoint functionality through runtime breakpoints.

  - A scalable and massively parallel *Runtime State Analytics Engine* on Apache Spark that achieves the construction and validation of *Runtime State Model*s from multiple simultaneous input sources, through efficient graph analytics. The engine supports the online construction of *Runtime State Model*s on streams of runtime events captured throughout the course of programs' executions.

- First to evaluate the effectiveness of using *Runtime State Model*s in the context of software anomaly detection. We introduce three types of injected faults to the RUBiS server, and show evidence that the *Runtime State Model*s could help detect runtime anomalies and provide useful information in fault localization.

1.2 Thesis Outline

This thesis is consisted of 7 chapters.

- In **Chapter 2**, we provide background knowledge that are relevent to the understanding of this thesis, including the formal definitions of *Runtime State*s and *Runtime State Transition*s and *Runtime State Model*s, along with brief introductions to the technologies used in the implementation of *Xtract*.

- In **Chapter 3**, we discuss related work that approached the problem of software fault and anomaly detection, with runtime traces, log messages, and system metrics.

- In **Chapter 4**, we present the design and implementation of our *Runtime Data Extraction Infrastructure* that works to extract the runtime information of applications directly from the Java Virtual Machines.

- In **Chapter 5**, we present the design and implementation of our scalable and massive parallel *Runtime State Analytics Engine*. We also describe a set of graph algorithms used by the engine to construct, validate and compare the *Runtime State Models*.

- In **Chapter 6**, we talk about our experiment setup and environment; discuss experiments conducted to evaluate the performance overhead of *Xtract*, and show how *Runtime State Model*s help us in detecting and locating faults injected to a RUBiS server.

- We conclude in **Chapter 7**, identify, and list issues in current state of the work as future work.

Chapter 2

Background

In this chapter, we formalize the definitions of *Runtime States*, *Runtime State Transitions*, and *Runtime State Models* used throughout this thesis. We also provide necessary background information that are relevant to the understanding of our approaches, and technologies used extensively in our implementations.

2.1 Terminologies and Definitions

To formalize the definition of a *Runtime State Model*, we first refer to Lamport's formal definition of a *computation* [36, 38],

> **Lamport's Definition of a Computation:** A *computation* is a sequence of *steps*, $< s, \alpha, t >$, where s and t are *states* and α is an *action*. A *behavior* is a sequence $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \xrightarrow{\alpha_3} \ldots$. The step $< s_i, \alpha_i, s_{i+1} >$ represents a transition from *state* s_i to *state* s_{i+1} that is performed by *action* α_i.

In the context of this thesis, we define a *state* as a set of (*variable, value*) pairs, a *step* as a transition of states induced by the change of a *variable's value*, and use a sequence of *variable's value* changes to define a *behavior*.

2.1.1 Runtime Trace

We define the sequence of *variable's value* changes as a *Runtime Trace*.

Definition: A **Runtime Trace** RT is a *sequence* $e_1, e_2, \ldots$ from contiguous observations $o_1, o_2, \ldots$, where each $e_i \in RT$ indicates a variable's value change event. Each change event $e = (v \rightarrow val_v)$ represents the value of variable v has been changed to val_v.

2.1.2 Runtime States and Transitions

*Runtime State*s and *Runtime State Transition*s are derived from a *Runtime Trace*. We derive our own formal definitions of *Runtime State*s, and *State Transition*s, based on Lamport's definition of a *computation*, as follows,

Definition: A **Runtime State** s_i is a *set* of variables with values, derived from a *Runtime Trace* $RT = e_1, \ldots, e_i$, in their temporal order. Each variable $(v = val_v)$ represents the variable v has a most recent value of val_v. A *Runtime State* without any variables is defined as the initial state s_0.

One *state* is transitioned to another *state* through a *state transition*, where

Definition: A **Runtime State Transition** $t_{src \rightarrow dst} = (s_{src}, s_{dst}, e)$ represents a transition from state s_{src} to state s_{dst}, given a variable value change event e.

2.1.3 Runtime State Models

A *Runtime State Model* is a *set* of *Runtime State*s *and* a *set* of *Runtime State Transition*s. We give the formalized definition of a *Runtime State Model* as follows,

Definition: Given a *set* of *Runtime State*s $S = s_0, s_1, \ldots s_n$, and a *set* of *Runtime State Transitions* $T = \{t_{i \rightarrow j} | s_i, s_j \in S, i \neq j\}$. A **Runtime State Model**, $G = (S', T')$, is a *set* of *Runtime State*s $S' = s'_0, s'_1, \ldots s'_n$, where $s'_i \subseteq s_i$, and a *set* of *Runtime State Transitions* $T' = \{t'_{i \rightarrow j} | s'_i, s'_j \in S', i \neq j\}$.

Each *Runtime State* in the model is reachable from each other, if there exists a *Runtime State Transition* between the states. A *Runtime State Model* is represented with a state machine, a directed graph with each vertex being a *Runtime State*, and each edge being a *Runtime State Transition*.

```
1 int simple_function() {
2   int a = 1;
3   int b = 2;
4   a = 3;
5 }
```

Figure 2.1: A Simple Function Generating 3 Change Events

2.1.4 Runtime State Pruning

Runtime State Pruning is the process of computing a set of *Runtime States* S', where each state $s' \in S'$ is a subset of the original states $s \in S$ derived from a *Runtime Trace*. We define the process of *Runtime State Pruning* as follows,

Definition: Given a set of variables $V = \bigcup v, \forall v \in s, \forall s \in S$, where $S = s_0, s_1, \ldots, s_n$ is the original *set* of *Runtime States*. Derive a set of states S', where $S' = s'_0, s'_1, \ldots, s'_n$, $s'_i \subseteq s_i, \forall i \in [0, n]$, such that $v' \in V'$, $\forall v' \in s', \forall s' \in S'$, where $V' \subseteq V$. We define the process of deriving S' as **Runtime State Pruning**.

2.1.5 Universal Runtime State Models

We define the common subgraph shared by multiple *Runtime State Models* as a *Universal Runtime State Model*. We formalize the definition of a *Universal Runtime State Model* as follows,

Definition: Given a set of *Runtime State Models*, $G_1 = (S_1, T_1), G_2 = (S_2, T_2), \ldots, G_n = (S_n, T_n)$. A **Universal Runtime State Model** $G_u = (S_u, T_u)$ is such that $s \in S_u$, iff, $s \in S_1 \cap \cdots \cap S_n$, $\forall s \in S_1 \cup \cdots \cup S_n$, and $T_u = \{t_{i \to j} | s_i, s_j \in S_u, i \neq j\}$

Given that a *Runtime State Model* summarizes the *behavior* of a program, a *Universal Runtime State Model* summarizes the *common behavior* of a set of executions.

2.1.6 Case Study

Take the example in Figure 2.1, at the time of observation, if the control is at source code line 3, we say the runtime state at observation o_i is $s_i = \{(a = 1)\}$ with a transition

$t_{i-1 \to i} = (s_{i-1}, s_i, (a \to 1))$. When the control reaches line 4, we say the runtime state at observation o_{i+1} is $s_{i+1} = \{(a = 1), (b = 2)\}$ with a transition $t_{i \to i+1} = (s_i, s_{i+1}, (b \to 2))$. When the control reaches line 5, and updates the value of a, we say the runtime state at observation o_{i+2} is $s_{i+2} = \{(a = 3), (b = 2)\}$ with a transition $t_{i+1 \to i+2} = (s_{i+1}, s_{i+2}, (a \to 3))$.

2.2 Java Virtual Machine Internals

This thesis focuses on the extraction and analysis of runtime information of Java applications, and therefore, requires the knowledge of basic internals of the Java Virtual Machine. In this section, we discuss and explain some of the mechanisms essential to the understanding of the rest of the thesis, in a nutshell.

2.2.1 Java Methods, JIT, and Debugging Support

Before executing a Java application, one needs to first compile Java source files using *javac*, which compiles Java code into JVM bytecodes, stored in class files. When starting a Java application, JVM will first load all classes into the VM, this process parses the class files, and stores methods in a special space in the heap, containing information of method types, method bytecodes, local variable tables, etc. Each method is identified by a unique ID associated with its enclosing class.

One interesting question that many may have is how is JIT, Java's Just in Time compilation going to impact the management of, and the subsequent interactions with Java methods. Java's JIT compiler aims at improving the performance of Java applications by optimizing, and eventually replacing Java's platform independent bytecodes to native machine instructions. Despite different JVM implementations, most adopt the hot code replacement strategy that only optimizes heavily used function through careful calculations due to the fact that the overhead of code compilation and replacement could be substantial [6].

Take Hotspot VM as an example, it offers both bytecode optimization and JIT compilation mechanisms that would progressively apply optimizations until eventually swapping bytecode to machine code [21]. While the replacement and compilation strategies are beyond the scope of this thesis, we will discuss how this affects the debugging of a Java application as follows.

A JVM is capable of providing full debug support while maintaining as many enabled optimizations as possible. Setting breakpoints as an example, the JVM will insert a breakpoint opcode at the corresponding instruction locations of the original bytecodes, and each version of the optimized bytecodes if any. Given that a bytecode may be optimized during execution, whenever a breakpoint is reached, the virtual machine would preserve all Java states, and temporarily fall back to use the original bytecode for debugging purposes. Since it is impossible to debug native machine code through the JVM interpreter, setting a breakpoint would disable further JIT compilations on the method until all breakpoints on the method are cleared. If a method is already JIT compiled at the point a breakpoint is being set, the method will be deoptimized [20], i.e., the JVM will fall back to use bytecode interpretation for that method without compromising program states with OSR (On Stack Replacement).

2.2.2 Java Local Variables and Bytecodes

Each Java method has a list of local variables, the number and length of which are determined and stored in the class files during compile time. A Java local variable typically has the following metadata, the notions of which will be used throughout the thesis,

- *slot*, the logical position of the variable in the list, used and identified by the virtual machine.
- *start_location*, the index of bytecode instructions where the local variable is first available.
- *length*, the length of the valid section for the local variable, i.e., the last bytecode instruction that this local variable is valid is $start\_location + length$.

Java local variables could be either a Java primitive, or a reference to an object that is stored in the Java heap. Accesses and Modifications to a Java local variable are translated to bytecode instructions to load or store values to a specific local variable slot. An example showing the mapping between the Java bytecodes and Java code is depicted in Figure 2.2, in which the upper left listing shows the original Java code that a method takes one argument, initializes and modifies two local variables. The Java code is compiled into a series of bytecodes as in the upper right listing that will be discussed later, and a list of local variables as shown in the table. Both bytecode instructions and the local variable table are extracted from the compiled class file using *javap*.

Although we will not go deep into Java bytecode instructions, we briefly discuss the instructions used here to explain how local variables are accessed by a Java program. There are three categories of instructions in the upper right list, where <*t*>*const*_<*i*> pushes a constant i of type t into the operand stack, <*t*>*store*_<*i*> stores a value of type t on the top of the operand stack back to the local variable at slot i, and <*t*>*load*_<*i*> load the value of type t at slot i to the operand stack. Note that if the slot number i too large to be represented by a single instruction, two instructions will be used with the second instruction being the slot number, as in lines 4-5 and lines 9-10. That said, we can now walk through the bytecodes as follows,

The interpreter first pushes integer 0 into the stack, and stores it to local variable at slot 3, i.e., *intVariable*, this corresponds to the Java code line 3. The same logic is followed by instructions 3-5 corresponding to Java code line 4, and instructions 6-7 corresponding to Java code line 6. For Java code line 7, one needs to first load the value of *doubleArgument* and store its value into variable *doubleVariable*, this corresponds to instructions 8-10.

The local variable table has the information of each local variable per method, including their slot number, start location, length, variable name and signature. In our example, 4 local variables exist in *someMethod*. Note that, for a member method, slot 0 is always reserved for *this* object. Since all of the local variables in this case are reachable until the end of the method, the last instruction that the variables are valid is $start\_location + length = 12$, though their *start_location*s vary. Since both *this* and *doubleArgument* are reachable since the entry of the method, their *start_location* are 0, while the *start_location* of *intVariable* and *doubleVariable* are 2 and 5 respectively.

2.3 JVM Tooling Interface

The JVM Tooling Interface (JVMTI) is a set of APIs provided by Java Virtual Machines to inspect the states and control the execution of Java applications [16].

To use the JVMTI interfaces, one needs to implement a client, namely agent. An agent is essentially a shared library written in C/C++, loaded by the JVM at startup time, and piggybacks on the JVM process. One could ask the JVM to load an agent by specifying the agent path and options in *JAVA_OPTS*[1].

The JVMTI provides control mechanisms that fall into various categories, however, we discuss only two of those that are used extensively in our implementation.

[1]The command-line option to load an agent is -agentpath:<*path-to-agent*>=<*agent-options*>

```
1  class LocalVariableExample {
2    public int someMethod(double doubleArgument) {
3      int intVariable = 0;
4      double doubleVariable = 0.0;
5
6      intVariable = 1;
7      doubleVariable = doubleArgument;
8
9      return 0;
10   }
11 }
```

```
iconst_0          1
istore_3          2
dconst_0          3
dstore            4
4                 5
iconst_1          6
istore_3          7
dload_1           8
dconst            9
4                 10
iconst_0          11
ireturn           12
```

start_location	length	slot	name	signature
0	12	0	this	LLocalVariableExample;
0	12	1	doubleArgument	D
2	10	3	intVariable	I
5	7	4	doubleVariable	D

Figure 2.2: An example of the Mapping between Java Bytecodes and Local Variables

2.3.1 Classes and Fields

A JVMTI agent has the capability to extract class information from the JVM. The *GetLoadedClasses* function returns all class objects from the JVM that have already been loaded. The *GetClassSignature* function gives the Java class signature of each class object. Together, one could easily implement logic to get a subset of useful classes according to the class signatures. A Java class usually defines fields and methods.

One could get a list of fields given a class object with *GetClassFields*. Java fields are uniquely identified as *jfieldID*s in JVMTI, and are valid until their enclosing classes are garbaged collected or modified. The names and signatures of Java class fields could be extracted with *GetFieldName*.

2.3.2 Methods, Bytecodes, Breakpoints and Local Variables

Similar to Java class fields, one could extract a list of methods given a class object with *GetClassMethods*. Java methods are identified with a unique ID of type jmethodID in JVMTI, with which one could get various information of the method, including the name, the signature, the modifier, etc. Among the miscellaneous stuff one could extract from the JVM, we discuss two important pieces to our implementation: the method bytecodes and local variables.

As mentioned in Section 2.2.1, the bytecodes and local variables of Java methods are generated and stored in the class files during compile time, and despite JVM's optimizations to the bytecodes, it will always fall back to use the original bytecodes if in debug mode. That said, *GetBytecodes* function returns an array of original bytecodes as compiled by *javac* given a *jmethodID*, with each element being a single bytecode instruction.

One could set breakpoints at various bytecode locations using *SetBreakpoint* that takes two parameters, the method id, and the index of the instruction in the array, at which to set the breakpoint. Setting breakpoints on a method immediately deoptimizes the method, and invalidates its capability to be JIT compiled until all breakpoints on the method are cleared. Events could be enabled on the breakpoints. If a breakpoint is reached, a breakpoint event will be triggered invoking a user-defined callback function. We discuss JVMTI events in Section 2.3.3

A mapping from bytecode locations back to the Java code locations could be obtained through *GetLineNumberTable*. One could also extract a list of local variables given a Java method. The *GetLocalVariableTable* function takes a method id, and returns a list of Java

local variables including the *slot*, *start_location*, *length*, along with the name and signature of each variable defined in the method.

2.3.3 Event Management

JVMTI provides mechanisms to manage various JVM events, e.g., FieldModifications, FieldAccesses, Exceptions, Breakpoints, etc.

Event notifications are triggered by invoking corresponding event callback functions. A global struct containing callback function pointers of all event types are set at the agent loading phase through *SetEventCallbacks*. This also indicates that only one callback logic per event is permitted. Notifications could be enabled or disabled for each event with the function *SetEventNotificationMode*, globally or with a per thread granularity.

In the scope of this thesis, two events are of interests. The field modification event that is triggered whenever a field is modified, and the breakpoint event that is triggered whenever a breakpoint is reached.

To set a modification watchpoint on a Java class field, one needs to call *SetFieldModificationWatch* providing the class object and the *jfieldID* of the field. If the watchpoint is no longer needed, *ClearFieldModificationWatch* could be called with the same set of arguments. A callback of the field modification event gives the following information,

- the thread that is modifying the field
- the method and instruction location that is modifying the field
- the class, signature, object of the field being modified
- the new value of the field

Similarly, to set a breakpoint, one need to call *SetBreakpoint* with the *jmethodID* and the location of instruction at which to set the breakpoint. Breakpoints could be cleared with *ClearBreakpoints* with the same set of arguments. The breakpoint event will be triggered before the execution of the instruction at which a breakpoint is set, and the callback of the breakpoint event gives information of the current thread, the method id and instruction location. The JVMTI and JNI environment pointers are provided in the breakpoint callback that could be used to access runtime information from the JVM. The thread will be temporarily suspended until the callback returns.

2.4 Apache Spark

Apache Spark is a batch processing system that claims to be up to 100x faster than Hadoop MapReduce in memory [53]. We use Spark extensively in our runtime data analysis and modelling process. In this section we briefly discuss various components of Spark that we adopt in our implementation.

2.4.1 Resilient Distributed Dataset (RDD)

Spark uses Resilient Distributed Dataset (RDD) to achieve fault-tolerant distributed in-memory batch processing [58].

An RDD is a read-only, partitioned collection of records. Unlike in Distributed Shared Memory (DSM) where processes are allowed to read and write to a particular address, RDDs only provide coarse grained transformations, i.e., map, filter, reduce, etc, that enable the tracking of how an RDD is derived from other RDDs, or from the file system by logging those transformations (lineage), which could be later used in system recoveries in the presence of failures. Other benefits of RDDs over DSM include the ease of migrating jobs from slow nodes, the capability of fine-grained scheduling based on data locality, etc., according to the original paper [58].

Operations on an RDD are divided into two categories, transformations and actions. A transformation is a lazy operation that define a new RDD, while an action launches a computation to calculate a value or write data to external storage. One RDD may be transformed multiple times before an action is applied. A summary of common RDD operations is shown in Table 2.1.

2.4.2 Spark Streaming and D-Streams

As an effort to support real-time streaming processing with Spark, discretized streams (D-Streams) is proposed to simulate real-time streaming with a batch processing framework [59].

A D-Stream groups input streams into a series of RDDs on small time intervals, and computes over the RDDs through batch processing. That said, one needs to first specify a window size. Input data received in each window is stored across the cluster to form an input dataset, which is manipulated by users through RDD operations that act independently on each window.

Category	Operation	Description
Transformations	*map*	transform an RDD from type A to B, one to one
	flatMap	transform an RDD from type A to B, one to many
	reduceByKey	reduce multiple elements with the same key
	groupByKey	group multiple elements with the same key
	filter	filter out elements that does not satisfy a predicate
	join	join two RDDs
	leftOuterJoin	perform leftOuterJoin with the other RDD
	rightOuterJoin	perform rightOuterJoin with the other RDD
Actions	*collect*	return an array containing all elements of the RDD
	count	return the number of elements in the RDD
	reduce	reduce all elements to one variable
	save	write RDD to an external storage system

Table 2.1: A Summary of Common RDD Operations

```
1 val streamingContext = new StreamingContext(..., Seconds(10))
2 val lines = streamingContext.socketTextStream("localhost", 7000)
3 val words = lines.flatMap(_.split(" "))
4 val wordCounts = word.map(x => (x, 1)).reduceByKey(_ + _)
5 wordCounts.print()
```

Figure 2.3: An Example of Network Word Counting with Spark Streaming

An example of counting the number of words in a network stream is shown in Figure 2.3[2]. In the example, a window size of 10 seconds is specified on line 1. By creating a network text streaming from the localhost, Spark Streaming would create RDDs with records received in 10-second windows, and apply 2 *map* and 1 *reduce* operations on those RDDs, giving the count of words received in each window.

A D-Stream inherits all benefits of an RDD, including fault tolerance through RDD lineage. To achieve efficient recovery, an approach called *parallel recovery* is introduced by periodically checkpointing some of the state RDDs, and asynchronously replicating them to other nodes. In the presence of a failure, multiple parallel tasks are launched to compute and recover different RDD partitions from the latest checkpoint.

[2]You can find the full code here: https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/NetworkWordCount.scala

2.4.3 Spark GraphX

Spark GraphX brings low-cost, fault-tolerant graph processing on a general-purpose data processing system, i.e., Apache Spark, that matches the performance of specialized graph processing systems [28]. Despite the great efforts made by Spark GraphX to optimize the representation, computation and partition of graph data, here we only discuss Spark GraphX from a user's perspective, i.e., the operators Spark GraphX provides to facilitate graph computations.

Spark GraphX provides a graph abstraction, namely **Graph[VD, ED]**, that has a vertex type of **VD** and an edge type of **ED**. It takes two *RDD*s representing the vertices and edges of a graph respectively. Each vertex of a *GraphX* graph is a key-value pair of **(VertexId, VD)**, while the **VD** could be any objects including user-defined structures. Each edge of a *GraphX* graph is in the form of a triple, **(src VertexId, dst VertexId, ED)**, in which the first two elements indicate the *src* and *dst* vertex ids, and the *ED* is an object describing the edge.

Spark GraphX supports both transformations on graph components, i.e., vertices and edges, as well as Pregel like operators. We provide a summary of GraphX operators in Table 2.2.

A Pregel-like operator implements a GAS message passing system with each vertex in the graph being an individual program [40]. In Spark GraphX, the Pregel implementation is implemented with batch processing operators that the user needs to provide four mandatory functions.

- *vprog* is used to simulate the vertex program that updates the state of each vertex according to incoming messages
- *send_message* is used to generate messages to send to each of the neighboring vertices. If no messages are to be sent to a vertex, an empty message set could be generated. The entire process terminates when the number of active messages becomes 0.
- *merge_message* is used to merge multiple messages into a single one before sending them out to a vertex. This is for performance considerations to reduce shuffling overhead.
- *initial_message* is the initial message to be sent to all vertices before the first super step. An initial message could be empty.

Operator	Description
mapTriplets	transform the graph type from [VD, ED] to [VD, ED'] given each triplet
mapVertices	transform the graph type from [VD, ED] to [VD', ED] given each vertex
mapEdges	transform the graph type from [VD, ED] to [VD, ED'] given each edge
subgraph	filter out vertices and edges that does not satisfy predicates
pregel	run pregel like impl with user-defined vertex programs and messages

Table 2.2: A Summary of GraphX Operators

A *triplet* in the *mapTriplets* function is a notion provided by GraphX that describes a triple of **(src vertex object, dst vertex object, edge)**. Different from an *edge* that only gives the ids of its src and dst vertices as provided, a *triplet* gives the actual objects of the three components.

We use Spark GraphX extensively in the building and analysis of software state machines as part of our software state modeling process.

Chapter 3

Related Work

In the context of runtime anomaly detection, existing research generally falls into three categories:

1. approaches to detect runtime anomalies and locate faults through the analysis of runtime traces, where a runtime trace could be an execution path, or a request flow between networked components. These approaches normally require instrumenting the programs for data extraction,

2. anomaly detection through log mining makes use of the pervasive log messages most software systems produce to determine whether, how, and when a failure happens inside of the system,

3. and approaches to detect runtime anomalies through system metrics, i.e., CPU utilization, memory consumption, etc.

3.1 Runtime Tracing and Models

Runtime tracing is the most intuitive approach when it comes to debugging. Existing research is observed to focus on two different aspects of this problem. Research that focuses on runtime tracing techniques proposes tools and implementations to extract runtime data to reveal a program's behavior. Efforts that focus on the automation of fault diagnosis using runtime traces usually construct models to look for behavior deviations. We discuss them separately

3.1.1 Runtime Tracing Techniques

Purify [30] uses static instrumentation to detect memory errors. It instruments C/C++ object files to add tracking code at each memory allocation and deallocation site. A memory error is reported if an address is accessed before allocation, or if an address is not properly deallocated.

As opposed to Purify, DTrace [11] uses dynamic instrumentations where each instrumentation probe could be enabled or disabled. It is built in the system kernel to track function invocations, syscalls, locks, etc.

Anderson et al. propose continuous sampling on the OS level with performance counters, and use an analysis tool to calculate time spent on each instruction, source code line, and procedure calls [2].

Bhatia et al. use fined-grained OS events, e.g., page allocations, process scheduling, block-level I/O, etc to detect system performance problems [7].

For the case of a networked system, approaches are proposed to track the request flows between distributed components.

X-Trace is a request tracing system for networked systems [24]. The idea is that each component of a networked system includes a metadata header in their message packets before sending them to the next hop, describing the identity of and operations conducted on the component. The system then analyzes the message flow and operation status of each component from a client on the receiving end to determine the failures of system components.

Dapper [51] is a production framework used inside of Google to trace RPC calls of a distributed system. It is a built into Google's RPC framework that keeps track of the source, destination, operation and timing information of each RPC call, and presents them in a tree structure. The traces are stored in an external log file to be analyzed by other entities.

These approaches provide means to extract runtime data, but they expect the developers to manually correlate the data to diagnose faults.

3.1.2 Expectations as a Model

Early efforts towards software fault diagnosis require manual input of expected program behaviors. These approaches usually employ special-purpose domain languages to describe an expectation.

Bates proposes the use of *Event-Based Behavioral Abstraction*, a model that uses events and attributes, e.g., for an open file event, the name and id of the file, to describe the behavior of a networked system.

Perl et al. use performance assertions to detect runtime performance anomalies. A developer needs to describe the expected performance metrics of various operations, e.g., I/O timing, lock wait time, cache hit rate, etc. Anomalies are reported if an operation fails the provided performance assertion [47].

Pip [49] is an infrastructure that detects unexpected behaviors in distributed systems. It requires programs to be linked against a library to generate events and resource measurements. It takes descriptions of the expected execution paths and performance metrics for each operation, and reports anomalies when mismatches are found.

3.1.3 Models from Performance Costs

Existing approaches that are designed to diagnose system performance issues usually construct cost models. Magpie [4] uses OS resource consumption events, e.g., bytes read, cache miss, etc., to establish a clustering model for each request.

Sambasivan et al. use request flows and the response time of each request captured by Dapper [51] to model the system performance in a directed weighted graph [50]. They use Kolmogorov-Smirnov test on response time and request counts to determine deviations in the request flows. The graph is used to identify anomalous flow ranked by the number of appearances.

X-ray [3] uses the execution paths and timing of each system call and synchronization operations to construct weighted directed graphs. The graph is then used to diagnose performance issues by looking at the edges with large weights.

These approaches focus on diagnosing system performance issues, and can not be applied the diagnosis of non-performance related issues.

3.1.4 Models from Paths and Flows

Chen et al. designed *Pinpoint*, an instrumented J2EE middleware that tracks the client requests, internal and external failures [14]. This results in a matrix with each rowing being a single request, and each feature being the number of failures happened for each of the software components. A clustering algorithm is later applied on the generated matrix

to determine the failing component. However, it is implementation-agnostic, and can only be used to determine the failing components.

For efforts to detect more generic types of faults, Ghanbari et al. came up with a low-overhead real-time solution to detect runtime anomalies, namely, Stage-aware Anomaly Detection (SAAD) [27]. SAAD looks for software logging points, i.e., code statements that print out log messages, through source code analysis. They instrument the source code to, instead of printing out a message to a log file, send an event message to a remote analyzer. Further analysis are carried out through statistical testing, i.e., t-test on the execution flows and time spent in between consecutive logging points for the detection of flow and performance anomalies. This approach, however, suffers from high false positives.

We recognize ClearView [46] as the most related work to ours. ClearView is an automatic error patching facility that corrects failing executions online by enforcing the execution paths. It constructs invariants, a model that captures the common control flows (a sequence of instructions and variable values) across multiple executions. Failures, for example, illegal control flow transfers and memory accesses, are detected by instrumented monitors. Whenever a failure is detected, ClearView repairs the actual execution by enforcing the flow recorded in the invariant. This is in contrast to our goal of using invariant runtime states to detect anomalies and locate faults.

3.2 Log Mining

Log messages are pervasive, and are intended to be used to detect problems in any large-scale software. Analyzing runtime logs is usually an offline process, and therefore does not exert extra overhead on existing systems. Existing log analysis techniques usually take the following three approaches,

- Using natural language processing techniques that treat log messages as unstructured data.
- Inferring the structures of log messages through code analysis and annotations.
- Combining the efforts of machine learning techniques with manual data labeling.

In this section, we discuss approaches that use software logs to determine runtime anomalies.

Xu et al. use the console logs of a software system to detect anomalies [55]. Their approach requires the use of source code to recover the underlying structures of the log messages, and apply machine learning and information retrieval techniques to detect unusual patterns in the logs.

SherLog [56] is a tool that infers information to help programmers understand what have happened during the failed execution. It uses both the software source code and the log messages to infer the execution path and variable values during the failed execution. They conclude that *SherLog* is useful in diagnosing 8 real world software failures.

Nagaraj et al. present *DISTALYZER*, an automated tool to support developer investigation of performance issues in distributed system, by comparing the system logs to infer the variable values and event occurrences that exhibit the largest divergence across the log sets [44]. It compares the runtime logs by looking at both *events*, i.e., operations, and *states*, i.e., value of some system variables, and reports those information where performance variations are observed.

lprof [60], is a profiling tool that automatically reconstructs the execution flow of each request in a distributed application. It analyzes the application's binary code and runtime logs to associate log messages with individual requests, i.e., identifying log messages on distributed nodes that belong to the same request, and shows diverging message patterns per request, thus helping developers to find bugs.

Despite the requirements of access to the source code of the above approaches, Reidemeister proposes treating log messages as unstructured text [48]. It clusters log message by tokenizing the log messages, and applying clustering based on the edit-distance between message tokens.

The effectiveness of log analysis approaches depends greatly on the quality of log messages, i.e., counting on the developers to provide useful logging statements in the source code. Work that automatically inserts or enhances existing logging statements [57], requires instrumenting the source code.

3.3 System Metrics Models

Another relevant category of work uses system metrics data, i.e., CPU utilizations, memory consumption, etc., to detect software anomalies. The use of system metrics achieves anomaly detection with complete ignorance of the implementations and structures of the software system under observation.

Jiang proposes the modeling of system metrics data with linear and information theoretic models [33]. An unhealthy system state is determined through the fitness of the trained model given a data point representing the system state. System metrics can also be used to determine faulty components through the analysis of per-component metrics data.

Munawar et al. present approaches to discover system faults with linear correlations between system metrics data [41, 42, 43]. A correlation model relates multiple system metrics, and is sensitive to the fluctuations. Anormaly detection is achieved through the identification of mismatches between metric observations and predictions with the correlation models built with metrics of a health system.

These approaches can only be used to determine the failing components, but will not provide information to aid debugging.

Chapter 4

Runtime Data Extraction Infrastructure

To construct runtime state models, and apply them in the context of software anomaly detection, we propose *Xtract*, a general-purpose facility that automatically retrieves runtime data from the Java Virtual Machines, and automates the process of constructing and analyzing runtime state models. We divide the facility into two components,

- A *Runtime Data Extraction Infrastructure* that retrieves runtime data directly from Java Virtual Machines, and exposes a set of APIs to the monitoring entity to control the types and granularities of data retrieved.
- A *Runtime State Analytics Engine* that constructs, validates and analyzes the runtime state models from simultaneous input sources.

We present the *Runtime Data Extraction Infrastructure* in this chapter.

The infrastructure, is designed to extract runtime information from the Java Virtual Machine. In addition to data extraction using JVMTI and JNI interfaces as described in Section 2.3, it also implements logic to organize, serialize, and transport those data out of the Java Virtual Machine, and exposes a set of APIs to allow the external entity to control its behaviors.

We recognize two major component in the Xtract infrastructure.

- The Xtract RPC Service is a foundational service that provides efficient inter and intra components data serialization and transport support.

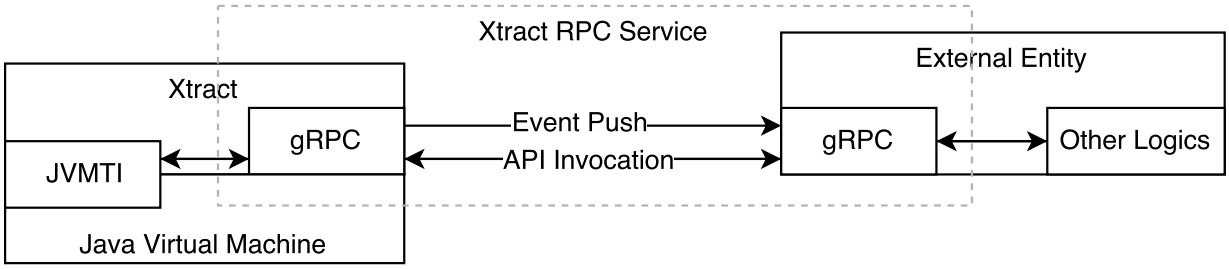


Figure 4.1: Xtract RPC Service Structure

- The Xtract JVMTI agent is a shared library that piggybacks on the Java Virtual Machine process. It exposes a set of higher level runtime inspection APIs to external entities through the RPC Service and makes use of the JVMTI and JNI interfaces to extract information from the JVM.

Our *Runtime Data Extraction Infrastructure* is implemented with 3,000 lines of C++ code for the JVMTI agent, 250 lines of Protocol Buffer definitions for the RPC service, and 300 lines of Go code. We describe each of the components separately as follows,

4.1 Xtract RPC Service

To accommodate communications between external entities and the infrastructure, a gRPC server is implemented in the shared library loaded by the JVM, and is initialized during the agent's *OnLoad* phase, as depicted in Figure 4.1. The goals of the RPC service are three folds:

1. External entities are able to get data out of the Java Virtual Machine on demand by calling corresponding interfaces.
2. The external entity could configure Xtract on the fly through a set of control APIs, e.g., to enable or disable an event or toggle the watchpoints or breakpoints during runtime.
3. The RPC service decouples the JVMTI agent implementations and data analytics logics in the external entities, adding to the flexibility and extensibility of *Xtract*.

The Xtract RPC Service implements two helpful mechanisms to reduce the overhead that may arise from a regular RPC implementation, namely, the asynchronous and streaming RPC.

4.1.1 Asynchronous and Streaming RPC

Consider the case of the *GetHeapObjects* function, which is supposed to send multiple objects back to the caller. Given the complexity of the application, there could be millions of objects to be sent back per request. A regular RPC implementation either sends back a single object, as in the *get_a_heap_object* function, or a list of all objects, as in the *get_heap_objects* function in Figure 4.2a, per call. Unfortunately, both work terribly for our case, due to non-trivial overhead of millions of remote function calls per request or the massive memory consumption to cache all objects in memory before sending them back to the caller in batch.

Our solution to this is to implement the streaming RPC that allows the callee to send responses back to the caller in a fashion similar to writing byte streams to a TCP connection, in this case however, writing objects one by one within a single RPC call. To terminate a streaming RPC call, the callee needs to write a terminator to the caller indicating the end of the response stream, and the caller needs to end the processing logics when a terminator is received. We show an example of Streaming RPC in Figure 4.2b.

Another case where regular RPC implementation does not play well is when a breakpoint callback needs to send an event notification to the callee, but has to be blocked until the return of the call, which also blocks the underlying Java application, slowing down its execution substantially. To solve this issue, we combine asynchronous RPC with streaming RPC that all events are first cached in a circular queue, which are later sent out using streaming RPCs with separate threads.

We use asynchronous and streaming RPC extensively in our RPC service to reduce memory footprint and runtime overhead.

4.1.2 Interface Designs

We notice that information retrieved from JVMTI could be abstract and fragmented. For example, three functions need to called separately to get all information we need for a method, including the method id, name, and bytecodes. To simplify our RPC interface design, we organize those JVMTI functions into higher level RPC calls according to their logical functionalities, and group each fragmented information into structured messages as the parameters and return values of our interfaces. For clarity, we take the *GetClassMethods* interface as an example.

As mentioned earlier, getting all information of a method required three separate JVMTI function calls as shown in Figure 4.3a, however, for the simplicity of our RPC

```
class RPCServer {
  Object get_a_heap_object(...) {
    // process an object
    return  /* processed object */;
  }

  Object* get_heap_objects(...) {
    Object* objs = new Object[VERY_LARGE_NUMBER];
    for (int i = 0; i < VERY_LARGE_NUMBER; ++i) {
      // process an object
      objs[i] =  /* processed object */;
    }
    return objs;
  }
};

void get_a_heap_object_client(...) {
  Object* objs = new Object[VERY_LARGE_NUMBER];
  for (int i = 0; i < VERY_LARGE_NUMBER; ++i) {
      objs[i] = get_a_heap_object(...);
  }
}

void get_heap_objects_client(...) {
  Object* objs = get_heap_objects(...);
}
```

(a) Regular RPCs

Figure 4.2: A Comparison of Regular to Asynchronous and Streaming RPCs

```
class RPCClient {
  void push_an_event_async(const Event& e) {
    circular_queue.push_back(e);
  }

  void event_callback(const Event& e) {
    push_an_event_async(e);
  }

  void push_events_streaming(..., Writer<Event> writer) {
    while (!circular_queue.empty()) {
      writer.write(circular_queue.front());
      circular_queue.pop_front();
    }
  }
};

class RPCServer {
  void push_events_streaming(..., Reader<Event> reader) {
    while (!reader.terminated()) {
      Event e = reader.read();
    }
  }
};
```

(b) Asynchronous Streaming RPC

Figure 4.2: A Comparison of Regular to Asynchronous and Streaming RPCs (cont.)

interfaces, we provide one *GetClassMethods* function that extracts all the information from the JVM and group them into a single structured message that contains the *id*s, *name*s and *bytecode*s of a method. That said, our *GetClassMethods* interface takes a request from the caller indicating the classes, of which, the methods are to be extracted. It then iterates through and calls *GetClassMethods* on each of the classes to get a list of method ids, and subsequently calls *GetMethodName* and *GetBytecodes* on each method of each class. Before sending those information back to the caller, it wraps them into a list of *JavaMethod* structs, each containing the method id, name, and bytecodes of each method of each class. We show an abstract implementation of this process in Figure 4.3b. Note that besides omitting trivial syntaxes, the abstract implementation also shades the use of Protocol Buffer and gRPC for clarity.

The Xtract RPC service is implemented with the gRPC and Protocol Buffer framework. To define a gRPC function stub, one needs to provide a function signature, including the function return type, function name, as well as RPC request and response message types, among which, those types must be defined as Protocol Buffer messages. A Protocol Buffer message, referred to as a *proto* in later sections, is a language-independent structured message defined in *.proto* files. One can specify fields in a proto message, each of which could be a primitive, another proto message, or a repetition of primitives or proto messages. The gRPC function stubs and proto messages could be later compiled into the source files of various languages, including C/C++, Java, and Golang, that are used throughout our system. We describe our core interfaces and protos in Appendix A.

It is worth noting that most Xtract RPC interfaces produce *XtractStatus* as the response. *XtractStatus* is a proto message that describes whether a request is successful, and the error code and error message in the presence of request failures, however, it is to be distinguished from the *Status* object returned by the gRPC framework. We briefly discuss this by first showing a compiled C++ function signature of *SetBreakpoints* as follows,

```
Status SetBreakpoints(ServerContext* context, const
    MultipleSetBreakpointParams* request, XtractStatus* response);
```

In the C++ function, the *XtractStatus* defined as the return type in Figure A.2a has been made one of the function parameters named *response*, while the return type of the function is *Status*. The *XtractStatus* is a custom proto message sent by Xtract to show any internal errors that may occur during invocations of the JVM APIs, while the *Status* object generated and returned by the gRPC framework indicates whether or not the request failed due to a network or gRPC framework failure. As a client of the Xtract RPC service, both of these objects need to be inspected in the presence of failures to reason the cause.

```
// 1. Get a list of methods given a class object
jvmtiError GetClassMethods(jvmtiEnv* env, jclass klass, jint*
    method_count_ptr, jmethodID** methods_ptr);

// 2. Get the name of a method given the method id acquired in step 1
jvmtiError GetMethodName(jvmtiEnv* env, jmethodID method, char** name_ptr,
    char** signature_ptr, char** generic_ptr);

// 3. Get the bytecodes of a method given the method id acquired in step 1
jvmtiError GetBytecodes(jvmtiEnv* env, jmethodID method, jint*
    bytecode_count_ptr, unsigned char** bytecodes_ptr);
```

(a) Three Steps towards the Details of a Java Method with JVMTI

```
struct JavaMethod {
  jmethodID method_id;
  std::string method_name;
  unsigned byte* bytecodes;

  JavaMethod(...) { /* parameterized constructor */ }
};  // structured message containing required information of a Java Method

int GetClassMethods(const std::vector<jclass>& klasses, std::vector<
    JavaMethod>* methods_info) {
  // local variables declared here
  for (auto& klass : klasses) {
    env->GetClassMethods(klass, ..., &method_ids);
    for (auto& method_id : method_ids) {
      env->GetMethodName(method_id, &name, ...)
      env->GetBytecodes(method_id, ..., &bytecodes)
      methods_info->emplace_back(method_id, name, bytecodes)
    }
  }
  // clean up here
}
```

(b) Xtract Abstraction of *GetClassMethods*

Figure 4.3: An Example of *GetClassMethods*

4.1.3 Case Study

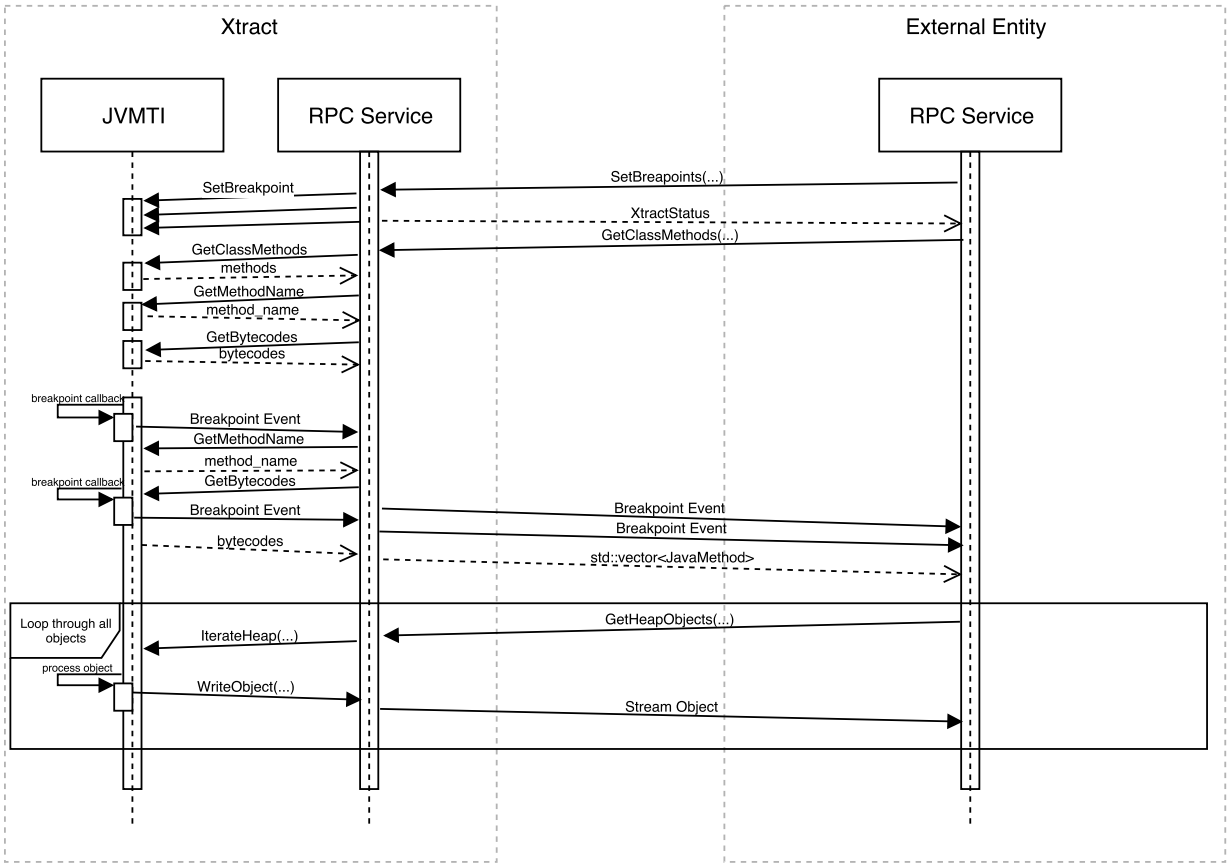


Figure 4.4: Sequence Diagram of the Xtract RPC Service, an Example

As a wrap up of this section, we present a sequence diagram of the Xtract RPC Service showing an example of asynchronous event pushing, object streaming, with an simultaneous invocation of *GetClassMethods* as in Figure 4.4.

Supposing the event management mechanism of this JVM is properly setup with the breakpoint event enabled. The external entity first sends a *SetBreakpoints* request through the RPC service, Xtract invokes corresponding *SetBreakpoint* function provided by JVMTI for each of the breakpoints specified in the RPC request. Upon return of the function, an XtractStatus indicating whether the function call is successful is return to the caller. Note that, with breakpoints set, JVMTI would now invoke breakpoint event callback before the execution of the instruction at which a breakpoint is set, which would send a event push

request to the RPC service, which would then send the breakpoint event to an external entity. After setting the breakpoints, the external entity now calls the *GetClassMethods* function.

Upon reception of the request, Xtract first gets a list of method ids for each class contained in the request. It will then get the name and bytecodes for each of those methods. After getting the bytecodes for the first method, JVMTI invokes the breakpoint callback which sends a breakpoint event to the RPC service. Note that the push operation is asynchronous that the callback function returns immediately without having to wait for the actual RPC call, which happens after the second *GetBytecodes* function in this case.

An event notification may also happen between consecutive JVMTI calls, as for the case of the second breakpoint event, however, these operations should never block each other with the exception that if the event happens to stop the world or the thread one may be trying to access. The RPC service organizes those information into a list of JavaMethod messages and send it back to the caller. Lastly we show the sequence of streaming RPC with the example of *GetHeapObject*. The caller sends a request to get heap objects from the JVM, for which the RPC service initializes a heap walk, and writes objects back to the stream one by one, whenever they are processed. Note that this is a simplified depiction of the process showing the work flow of a streaming RPC, while the implementation details are discussed in Section 4.2.

4.2 Xtract JVMTI agent

The Xtract JVMTI agent implements the JVMTI APIs and is compiled as a shared library that could be loaded by a Java Virtual Machine. Backgrounds in terms of what is, and how to use JVMTI is discussed in Section 2.3, however, in this section, we discuss how Xtract retrieves runtime data from the JVM with JVMTI and JNI, including, breakpoints resolution and stack local variables inspection.

We discuss each of these implementations in the following sections.

4.2.1 Breakpoints Resolution

A breakpoint is set on a method instruction. The thread will be paused before the execution of the instruction at which a breakpoint is set, and an event notification will be sent through an event callback function that reports the id of the thread, and the location of

the instruction that the thread is about to execute. To set a breakpoint, a call to the *SetBreakpoint* function needs to be made with the id of the method and the location of the instruction to set the breakpoint.

Setting a breakpoint with Xtract involves the invocation of two interfaces, *GetClassMethods* and *SetBreakpoint*. The first interface gets back a list of class methods, including their bytecodes, while the second interface sets a JVMTI breakpoint on a method bytecode instruction location. This way, we separate the determination of where to set a breakpoint entirely from the *Xtract agent*, and make it a decision of the external entity.

One breakpoint event notification sends one corresponding *JvmEventNotification* message to the external entity. The message contains the timestamp when the breakpoint callback is invoked, thread id, the name and id of the method, the name of the class that defines the method, and the location of the instruction at which the breakpoint event is triggered.

The message also includes a list of local variables available at the point of execution, used to derive local variable changes, which we now describe in the next section.

4.2.2 Stack Local Variables Inspection

Although desirable, the current latest version of JVMTI (JVMTI 1.2) does not provide approaches to set watchpoints on local variables [16, 25].

As a workaround, we implement Java local variable watchpoints with breakpoints, on instructions that change the values of local variables that we refer to as key instructions. For our case, we focus on the use of *istore*, *lstore*, *fstore*, *dstore*, and *astore*, that change the values of *integer*, *long*, *float*, *double* primitives, and object references respectively.

Note that, each of the instructions is presented in two possible forms.

- The *short form* uses a single instruction to access a value to a local variable slot, however, it only supports accessing values to local variable slot 0-3.
- Any operations that accesses the value of a local variable in slot greater than 3 adopts the *long form* that is expressed using two instructions with the first instruction being the op code, and the second being the slot number.

Breakpoints events are notified before the execution of an instruction, therefore, to get an event notification after a variable has been changed, a breakpoint should be set at

location $i + 1$ if it is a *short form* instruction, or at $i + 2$ if it is a *long form* instruction, where i is the location of the aforementioned instructions.

Recall that local variables are represented in the JVM in the form of tables, in which, each local variable is associated with *slot*, *start_location* and *length*. We determine whether a local variable is available at the point of execution by comparing the current instruction location as reported by the breakpoint callback, with its *start_location* and *start_location + length*, i.e., a local variable is determined as valid at instruction location i, iff, *start_location* $\leq i \leq$ *start_location + length*.

As an effort to reduce the runtime complexity of breakpoint callbacks, and potentially optimize the perceived performace, we return all currently available local variables, at the point of the event callback, back to the caller, instead of locating the one that was changed by the execution of the instruction. The caller can then figure out the variables being changed through a series of ordered breakpoint events without having to interfere with the execution of the Java application.

A breakpoint callback returns a *JvmEventNotification* message to the external entity. Apart from contextual information as described in the previous section, a breakpoint event message also contains a repetition of *JavaLocalVariable* messages, where a *JavaLocalVariable* message describes the name, signature and value of a Java local variable.

We now describe the process of retrieving stack local Java primitives and objects from the JVM as follows,

Retrieving Java Primitives

The retrieval of stack local Java primitives is achieved by first identifying the the types of the local variables. It is recognized that all primitive types in the Java Virtual Machines are associated with one-character signatures. i.e.,

- **Z** identifies a Boolean type,
- **B** identifies a Byte type,
- **S** identifies a Short type,
- **I** identifies an Integer type,
- **C** identifies a Char type,

- **J** identifies a Long Type,
- **D** identifies a Double type,
- **F** identifies a Float type

Four interfaces are provided to retrieve the values of Java primitives by JVMTI, namely, *GetLocalLong, GetLocalDouble, GetLocalFloat, GetLocalInt*. Note that the values of Boolean, Byte, Short, Integer, and Char typed variables should all be retrieved with the *GetLocalInt* interface.

In our implementation, we traverse the local variable table, and look at the types of variables. If a variable has a primitive type, we retrieve its value with these four interfaces.

Retrieving Java Objects

In addition to primitives, we observe that strings and objects are also important in the derivation of runtime states.

We use the signatures of the variables to determine their types. Variables that are not primitives, are recognized as Java objects. We distinguish Java strings from generic Java objects according to their signatures, that Java strings possess a signature of *Ljava/lang/string;*.

Stack local objects could be extracted with the *GetLocalObject* interface, which returns a reference of Java objects in the JVM heap in the type of *jobject*. If the Java object has a value of *null*, the returned *jobject* will be a *nullptr* in the C++ code.

If a Java object is a Java string, we use the JNI function, *GetStringUTFChars*, to retrieve the content of the Java string. We identify that the identifications of Java objects, i.e., their hash codes, are not a useful type of information in the construction of runtime states, however, we are interested in knowing whether the Java objects hold a null value. For Java objects, therefore, we only indicate if they are null. For null Java strings, we treat them as null objects.

4.3 Performance Implications

To conclude this chapter, we briefly discuss the performance implications of Xtract on the underlying Java application.

It is not difficult to see that JVMTI APIs are expensive, and may stall the execution of the underlying application substantially. Examples are, as discussed in this chapter, heap walks require the JVM to stop the world until all objects are visited, breakpoint events pauses the execution of the thread until the event callback is returned, and setting breakpoints on methods forces JVM to deoptimize the method, not to mention the additional overhead caused by event callbacks.

Many optimizations have been built into our implementations, including using asynchronous streaming RPC to handle event notifications, and implementing mechanisms to toggle watchpoints, breakpoints and control the granularity of event notifications on the fly, etc. However, this thesis focuses on approaches to extract and use runtime data to find program anomalies, where efficiency is not a major concern in our case.

Despite the fact, we believe further optimizations are feasible and are a matter of engineering efforts. We provide some insights here as follows,

- One could easily implement adaptive monitoring mechanisms that automatically refine the granularity of event notifications and disable unnecessary watchpoints and breakpoints through the Control APIs of the Xtract RPC Service on the fly.
- To avoid excessive breakpoints and watchpoints, it is possible to use code analysis techniques to annotate insignificant fields and variables to reduce the overhead.
- Further, performance impact could be avoided using the *Record and Replay* techniques, i.e., recording requests to the system and replaying them offline for anomaly detection.

We evaluate the performance impact of *Xtract* in Section 6.3.

Chapter 5

Runtime State Analytics Engine

We present our Runtime State Analytics Engine in this chapter. A runtime state, as described in Section 2.1, is a collection of key variables and their values representing the program's behaviors. A transition between two states is defined by the change of value of a variable.

Over the course of monitoring, the engine constructs a *Runtime State Model* for each individual program under observation, given the runtime data retrieved from our *Runtime Data Extraction Infrastructure*. The summarization of runtime states is conducted through a validation process that generates a *Universal Runtime State Model* across different executions of the same application. A runtime state comparison is then carried out to analyze anomalies through state deviations.

The engine is capable of computing and analyzing the runtime states from multiple incoming sources in massive parallel with distributed nodes, and is therefore both efficient and scalable. However, we assume that all incoming change events belong to the same application.

Our *Runtime State Analytics Engine* is implemented with Apache Spark, HDFS and ZooKeeper in 2,000 lines of Scala code, 300 lines of Java code and 200 lines of Go code.

We describe the system and the algorithms separately.

5.1 System Overview

We show the structure of our modeling engine in Figure 5.1. It consists of four components *Runtime Data Preprocessor*, *Runtime State Generator*, *Validator*, and *Comparator*, working

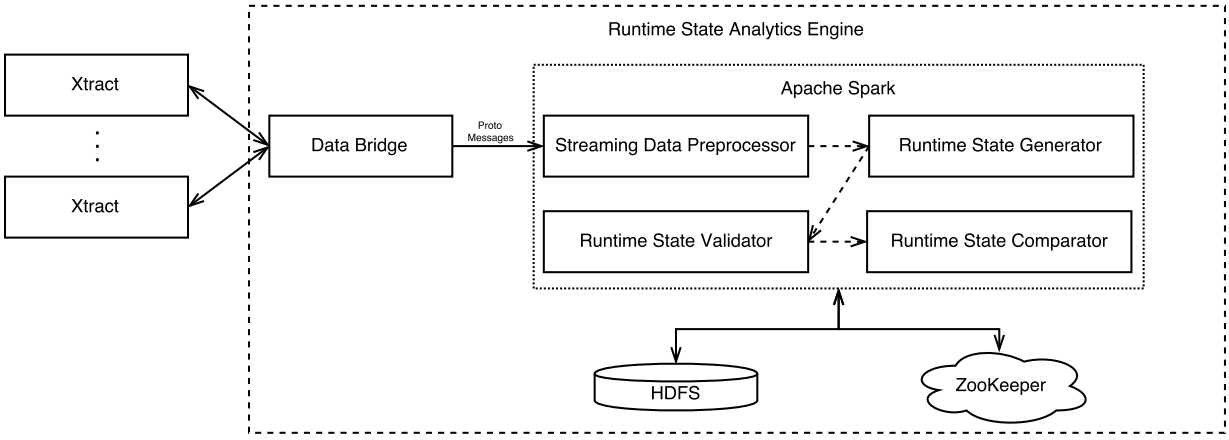


Figure 5.1: System Structure of the Runtime State Analytics Engine

in a pipeline fashion.

The engine takes input data directly from our runtime data extraction infrastructure, in real time. Data taken from the infrastructure is in the form of a stream, which is first segmented into multiple windows given a pre-defined window size. Each window is preprocessed through our *Data Preprocessor*, which produces intermediate representations of those input windows.

The intermediate representations of the input are consumed by the *Runtime State Generator* that generates a runtime state model, per input window, per source. A runtime state model is essentially a state machine, in the form of a directed graph with each vertex being a runtime state, and each edge being a transition between two states. The *Generator* is stateful, and implements online algorithms to gradually construct and refine the runtime state models according to the windowed runtime data input retrieved throughout the course of the programs' executions.

The *Runtime State Validator* summarizes the universal behavior of an application, i.e., common runtime states and transitions among multiple state models, while the *Comparator* analyzes two given state machines for behavior deviations.

We build the engine entirely on top of Apache Spark, making it both efficient and scalable. Communications between the components are fulfilled through HDFS and ZooKeeper, where the HDFS is used to buffer and store both temporary and output data, and ZooKeeper is used to implement a distributed producer consumer queue.

To connect the two entities, we implement a bridging facility, named as *Data Bridge* in the diagram, which implements a Google RPC server that receives event data from

the infrastructure, and streams the events through a network socket directly to the *Data Preprocessor*. We argue that decoupling the Modeling engine from Xtract infrastructure contributes to the flexibility and maintainability of the two separate entities. Note that, the engine supports simultaneous input from multiple sources that are processed in batch.

We describe each of the components as follows.

5.1.1 Streaming Data Preprocessor

Our streaming data preprocessor implements the Spark Streaming interface. It takes a stream of input data, which in our case, a stream of **JvmEventNotification** protos from the Xtract infrastructure, and apply batch processing on the input stream divided by a user-defined window, that is, using data received during the window as the input for subsequent batch processing logics.

To write a Spark Streaming application, one first needs to implement a receiver to receive input data as a discretized stream (DStream). A Spark Streaming receiver is essentially a network client that connects to a specified server, receives data and stores the data in the form of RDDs. Various general-purpose receivers are shipped with Spark Streaming, including the naive **socketTextStream** that receives a stream of text strings through network sockets, however, we implement our custom receiver that receives serialized proto messages from *Data Bridge*, and deserializes the messages before storing it to the underlying RDD. This enables the direct reusing of **JvmEventNotification** proto messages from the Xtract infrastructure without data translations between the two entities.

During the data preprocessing phase, we apply basic data transformations, e.g., sorting and filtering, that are supposed to be very fast with the Spark batch processing. Since Spark Streaming is essentially batch processing on windowed streams, doing so ensures that the processing of each windowed input could be finished during that window slice without having to stall the processing of subsequent windows. We describe our logic to preprocess those events in Section 5.2.

After the processing of each windowed input, we store the output RDDs to the HDFS as object files. These files are essentially a serialized representation of the RDDs, and are consumed by the *Runtime State Generator*. All files are named as **DataSignature_timestamp** for future indexing. We store the intermediate output to an external storage system for two considerations,

1. The construction of *Runtime State Model*s could be time consuming, using an external storage system as buffer reduces the amount of data that would otherwise build

up in the main memory.

2. Storing intermediate representations to an external storage system makes it possible to revisit historic data.

The *Runtime State Generator* runs asynchronously with the *Data Preprocessor*, therefore, to notify that an input has been processed, and is ready for further analysis, we also implement a distributed Producer-Consumer queue with ZooKeeper. The distributed queue records the paths of the object files stored by the *Data Preprocessor*, that are used by the *Runtime State Generator* to locate the object files. Whenever a new output is ready, the producer, i.e., the *Data Preprocessor* pushes the file paths to the queue, while the consumer, i.e., the *Runtime State Generator* receives a notification and reads the files. Records in the queue are persistent that they will persist in the ZooKeeper queue until consumed even in the presence of producer and consumer failures.

Input from multiple sources are processed simultaneously in batch, using the same set of algorithms. They are distinguished through an id to uniquely identify the source.

5.1.2 Runtime State Generator

The *Runtime State Generator* consumes the intermediate output from the *Data Preprocessor*, and produces a state machine representing the cumulative runtime states, for each of the input sources.

The generator starts by retrieving the file paths of intermediate output generated by the *Data Preprocessor* and loading those object files from HDFS as a set of RDDs. If the ZooKeeper queue is empty, it will block until an item is pushed into the queue by the preprocessor.

The generator will then build a state machine, i.e., a directed graph with each vertex showing a state of the program at an observation, and each edge showing a change of value of a variable. In our case, each change of value of the variables are captured by the *Xtract agent* through event notifications, therefore, we build the state machine according to the sorted change events by their timestamps.

It is non-trivial to build and access the state machines efficiently, especially when their sizes become large. Therefore, we compute, store and transform the state machines with Spark GraphX using distributed nodes, in parallel. We describe the process to build a state machine in Section 5.3.1.

One potential side effect of our approach is the state explosion, i.e., too many states and transitions due to excessive value changes of the variables. To address this issue, we select a portion of key variables from the variable collection to build the state machine, and implement mechanisms to transform the state machines to exclude selected variables, so that we ensure the compactness of the state machines to include only useful information. We describe our variable selection scheme and the state machine transformation mechanism in Section 5.3.3 and Section 5.3.4 respectively.

Note that, we build one state machine per window produced by the *Data Preprocessor*, however, the *Data Preprocessor* is a stream processor that produces output per pre-defined window. To keep track of the cumulative view of the program state, each state machine constructed from on a windowed input are merged into one global state machine. We describe the merging logics in Section 5.3.2.

If the input contains multiple sources, we keep track of intermediate states of each individual source separately. One state machine is generated per windowed input per source.

5.1.3 Runtime State Validator

Graph computations are expensive and time consuming. To speed up this process, we implement the runtime state validation process as a separate component that runs in parallel to the state generator, in a pipeline fashion.

The *Validator* takes the generated state machines, typically those generated from the same input window, and summarizes a common state to reveal the universal behavior of the application under observation. We describe the validation process in Section 5.3.5.

5.1.4 Runtime State Comparator

The *Runtime State Comparator* reads the state machines from HDFS and calculates the differences between two given state machines. It is used to derive the anomalous program states in our evaluations, where an anomalous state is defined as a deviation of runtime states and transitions in an execution with failure, from a healthy one.

We use Spark GraphX's batch processing operators to compare the state machines in parallel.

5.2 Runtime Events Preprocessing

Incoming runtime events are first preprocessed through a series of statistical computations and transformations.

A runtime event is defined as a pen-tuple of $e = (src, ts, t, v, val_v)$, where src indicates the source of the runtime event, e.g., the id of a process or the host name of a machine, ts indicates the timestamp when the change happens, t shows the thread where the event happens, v is the name that uniquely identifies a variable, and val_v is the value of the variable. Note that, as mentioned in Section 4.2.2, these incoming events from the *Xtract* infrastructure are not necessarily actual representations of variable changes, but rather, an indication of the value of each variable at the time of observation, for performance considerations.

During the preprocessing phase, we derive the following information according to the runtime events,

- A *set* of variables per thread per source, i.e., $V_{src,t} = v_0, v_1, \ldots$.
- The domain of each variable per thread per source, i.e., the variable's value set, $VAL_{src,t,v} = val_1, val_2, \ldots$.
- A collection of ordered changes of each variable per thread per source, i.e., $VC_{src,t,v} = (v \rightarrow val_{ts_0}), (v \rightarrow val_{ts_1}), \ldots$, where $val_i \neq val_{i+1}, \exists v \in V_{src,t}$, and $ts_j < ts_{j+1}$.
- A sequence of variable changes per thread per source, i.e., $CS_{src,t} = (v_0 \rightarrow val_{v_0}), (v_1 \rightarrow val_{v_1}), \ldots$, where $\exists v_i \in V_{src,t}$, and $v_i = v_{i+1}$, iff $val_{v_i} \neq val_{v_{i+1}}$.

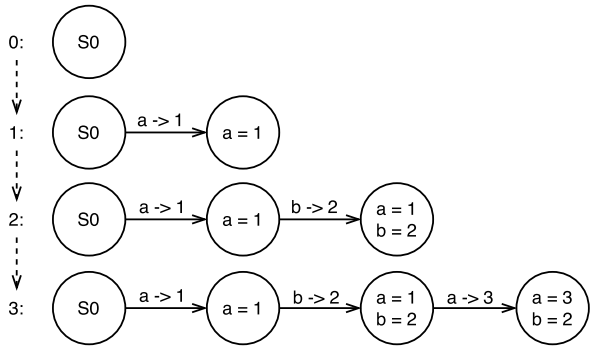
These information are derived from incoming events segmented by a pre-defined window. The processing of events of each window is stateless, and therefore does not require the knowledge of data in previous windows.

5.3 Runtime State Modeling

Constructing *Runtime State Model*s is essentially the process of building state machines indicating the state of program executions at each observation, and the transitions between states. Since the values changes of each variable is derived during the data preprocessing phase, we treat each variable value change as a state transition, denoted as a quadruple of $c = (ts, tid, v, val_v)$, where,

```
1 int simple_function() {
2   int a = 1;
3   int b = 2;
4   a = 3;
5 }
```

(a) Simple Function Generating 3 Change Events



(b) Building State Machines Sequentially

Figure 5.2: An Example of State Machine Construction

- **ts** is the timestamp when the change happens,
- **tid** is the id of the thread that changes the variable,
- **v** is the name of the variable, distinguished by the name of the variable, the name of the method where the variable is declared, and the name of the class that defines the method,
- and val_v is the new value of the variable in a string.

We now present our process to build a state machine according to variable change events.

5.3.1 State Machine Construction

Building a state machine really is a sequential procedure by gradually building a transition graph from a collection of change events in their temporal order. We start by walking through a simple example in Figure 5.2.

A simple code snippet shown in Figure 5.2a generates the following three variable change events, in order,

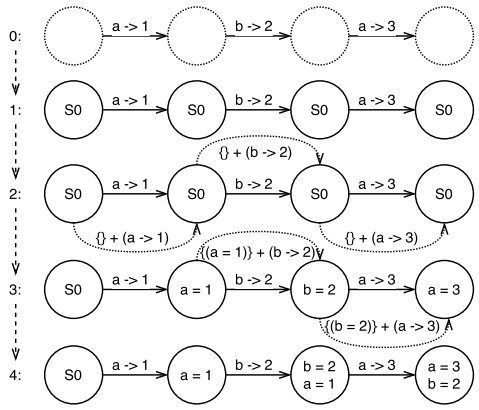
$$C = \{(0, 0, a, 1), (1, 0, b, 2), (2, 0, a, 3)\}$$

```
// 1. generate edges from change
   events
val edges = E.map(e => Edge(src, dst,
   t))

// 2. generate graph from edges
val graph = Graph({}, edges, S0)

// 3. populate graph
val graph_populated = graph.pregel(
  initial_message = {},
  send_message = e.src.t + e.t,
  vertex_merge = v.s  -> v.s + msg.s)
```

(a) Pseudo-code of Building State Machines with GraphX



(b) Steps of Building State Machines with GraphX

Figure 5.3: Building State Machines with GraphX

To build a state machine sequentially, we process the change events one by one while keeping track of the cumulative runtime state. As shown in Figure 5.2b,

1. the first event gives a transition from an initial empty state $s_0 = \{\}$ to $s_1 = \{(a = 1)\}$.
2. It is later changed by the second event to another state $s_2 = \{(a = 1), (b = 2)\}$, through a transition of $t_{1\to2} = (b \to 2)$.
3. Eventually, we get to the final state $s_3 = \{(a = 3), (b = 2)\}$ through $t_{2\to3} = (a \to 3)$.

We say variable changes $\mathbf{C}$ gives a state machine of

$$G = \{\{s_0, s_1, s_2, s_3\}, \{t_{0\to1}, t_{1\to2}, t_{2\to3}\}\}$$

It is trivial to build a state machine sequentially, however, we use Spark GraphX to efficiently compute, store and transform runtime state machines with distributed nodes. This is extremely helpful when the number of states becomes too large for a sequential implementation to manage. As a prerequisite of building a state machine with Spark GraphX, identical states need to be identified as the same vertex in a graph. We use the hash code of the variable map to uniquely identify the states.

To build a state machine with Spark GraphX,

1. we first generate a set of *edges* that uniquely represent a state transition from *src* to *dst* state. During this phase, we only need to uniquely specify the states with ids without having to create the actual state structures.

2. initialize a graph structure with empty vertex set, the *edges* we generated in step 1, and default state *S0* that is essentially an empty map. During this phase, GraphX would create a vertex structure for each vertex referenced by the edges. If a vertex does not exist in the provided vertex set, e.g., an empty vertex set as in our case, it creates a vertex with the provided default vertex, e.g., S0 in our case.

3. populate the graph in parallel with GraphX pregel implementation,

   (a) for each transition edge e, send a message from *src* to *dst*, if the state of *dst* is not equivalent to the state of *src* merged with the transition of e. The message contains a delta state $\Delta s = e.src.s + e.t$, that is the previous state, i.e., the state of the *src* vertex of each transition edge e, merged with the transition t, i.e., a variable's value change.

   (b) upon receiving a message from neighboring vertices, each vertex updates their own states with the delta state in the message, i.e., update their own states $v.s = v.s + \Delta s$, by merging them with the delta state.

   (c) keep looping from (3.a) until equilibrium is reached, i.e., no messages are sent from any vertex.

We show an example of building the same state machine with GraphX in Figure 5.3. Note that, the vertices in dashed lines indicate that they are not materialized until GraphX initializes the vertex structures with the default state S0. Equilibrium is reached after two iterations of message passing.

We argue that it is important to compute the state transitions on a per thread basis, given that the change events are essentially interleaved given the multi-threaded property of most applications, and that despite each thread may have their own state transitions, it is safe to later merge the per-thread state transitions into a single state machine.

The computation of state transitions per thread is sequential similar to the sequential state machine building process, with the exception that we only generate the edges, and compute the hash code of the previous and next states per change event. However, multiple computations are carried out in parallel with Spark. We merge the edges per thread into a single set before subsequent state machine building processes.

5.3.2 Temporal Join and Multi-thread Correlation

Merging multiple state machines is non-trivial.

Recall that we represent the state machines with GraphX as a directed graph $G = (S, T)$ with each vertex s identified by a unique id, and each edge t containing the id of its source and destination vertices. Therefore, merging multiple state machines $G_1 = (S_1, T_1)$, $G_2 = (S_2, T_2)$, ..., $G_n = (S_n, T_n)$ is essentially creating a directed graph with vertices and edge being the super set of each respective machine, i.e., constructing $G_s = (S_s, T_s)$, where, $S_s = \bigcup_{i=1}^{n} S_i$, and $T_s = \bigcup_{i=1}^{n} T_i$.

However, it is incorrect to simply merge the state machines constructed on each input window, due to the fact that a trace of a thread could be broken into multiple windows, and that the state machine constructed on a broken window is not a complete representation of the actual state transitions.

Consider the following example that separates the change events into three windows,

$$
\begin{aligned}
C_1 &= \{(0, 0, a, 1), (1, 1, a, 1), (2, 1, b, 2), (3, 1, a, 3)\} \\
C_2 &= \{(4, 0, b, 2), (5, 2, a, 1), (6, 2, b, 2)\} \\
C_3 &= \{(7, 0, a, 3), (8, 2, a, 3)\}
\end{aligned}
$$

We have three threads with **tid**s of 1, 2 and 3, executing the function in Figure 5.2a. Thread 1 finishes executing within the first window, thread 2 starts within the second window, and finishes within the third window, while the execution of thread 3 spans all three windows.

Without a proper merging process, our *Runtime State Generator* would generate three independent state machines labeled as *E1, E2,* and *E3*, and eventually merge them into a state machine labeled as *Merged* in Figure 5.4. It, however, does not reflect the actual state transitions of the program's execution, e.g., an execution of the program in Figure 5.2a will never change from an initial state directly to $s = \{(b = 2)\}$, or $s = \{(a = 3)\}$.

To merge windowed runtime events for a multi-threaded application, we recognize that two techniques could be applied in this case,

- *Temporal Join* that combines runtime events in two separated windows according to their temporal order.
- *Multi-thread Correlation* that groups runtime events according to their thread id.

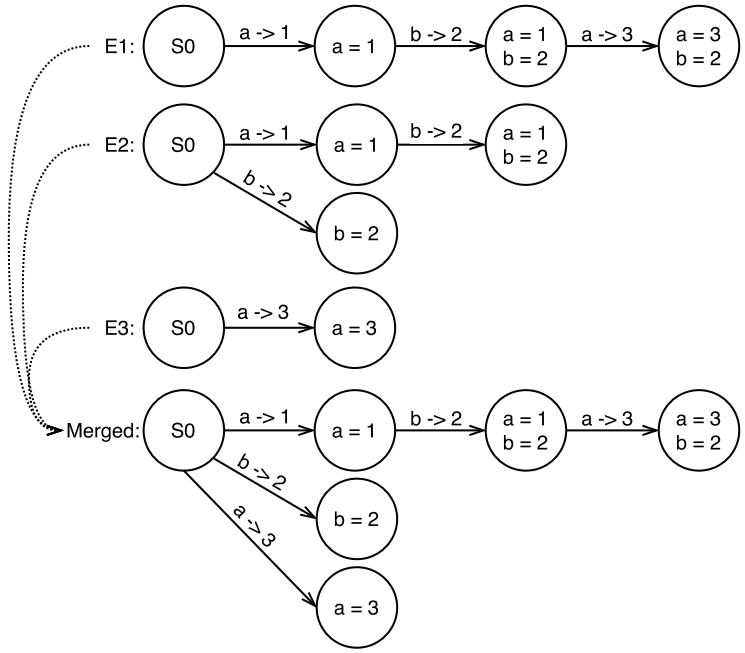


Figure 5.4: An Example of Incorrect Merging of State Machines

To employ the two techniques in the construction of *Runtime State Model*s, we present the *Cumulative Sliding Window* approach that keeps track of incomplete change events given consecutive change event windows.

We show a case of using the *Cumulative Sliding Window* approach to properly merge the state machines using the previous example. There are three windowed change events consisting of the executions of a simple program listed in Figure 5.2a, with 3 threads, identified as **tid** 1, 2, and 3. We depict the change events in three windows in Figure 5.5a, in which, *C1, C2, and C3* are the three windows corresponding to the three variable change sequences C_1, C_2, and C_3 above. Each line in the windows represents a sequence of change events in their temporal order per thread as recorded in a change event window.

With a *Cumulative Sliding Window (CSW)* of twice the size of a change event window, we keep track of incomplete change events as follows,

1. starting from the second incoming change event window, we compute a *CSW* to include events whose **tid** either presents in both windows, or presents in the second but first window, such that,

- if a **tid** is found in both windows, we include change events in *both* windows of that thread in the *CSW*
- otherwise, we include change events in the second window in the *CSW*

In our case, it is implemented with a single *rightOuterJoin* operation on the two window RDDs.

2. use the computed *CSW*, instead of the second window, for subsequent computations.
3. for a third incoming change event window, use the *CSW* as the previous window, and repeat from step 1.

We show merging the state machines with *Cumulative Sliding Window*s gives the correct results in Figure 5.5b. Note that merging two graphs uses only one of the duplicated vertices. We ensure that at most one edge exists between any two vertices, i.e., there exists at most one transition between any two states.

To correctly construct a *Runtime State Machine* with our *Cumulative Sliding Window* approach, three assumptions need to be made, which we describe as follows,

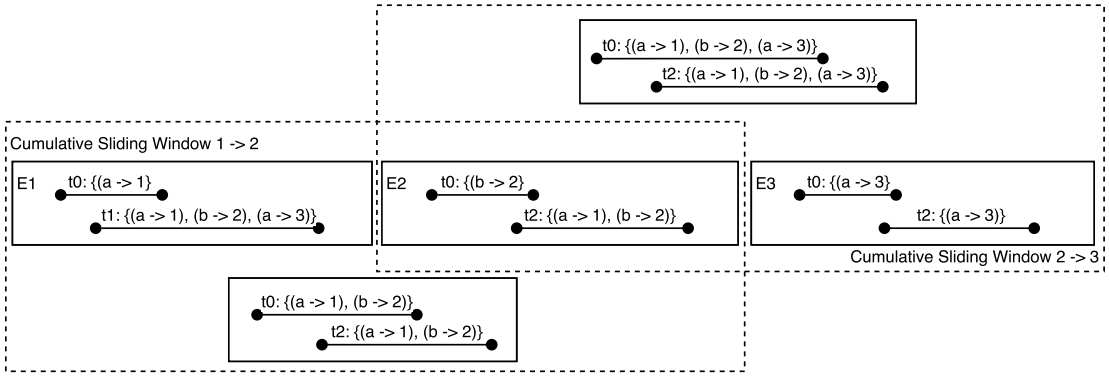
1. A thread could be uniquely identified with *tid*.
2. The same *tid* is not reused in two consecutive windows.
3. Events belonging to the same thread need to appear in any two consecutive windows.

We argue that these three assumptions could be fulfilled, where a *tid* is usually a monotonically increasing integer, while the second and third assumption could be fulfilled by adjusting the size of the windows.

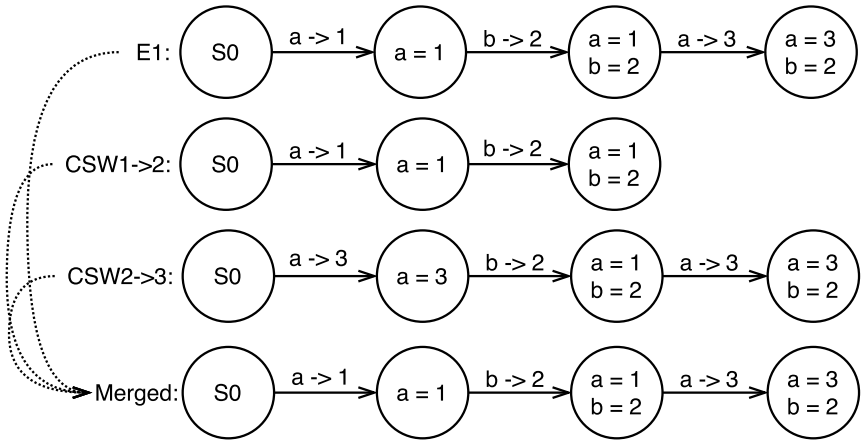
5.3.3 Variable Selection

It is observed that a *Runtime State Model* constructed from a runtime trace may contain too many states due to excessive value changes of some variables, e.g., timestamps, unique identifiers, etc. A *Runtime State Machine* containing too many states is noisy, and ineffective in the detection of anomalous states. We argue that variables with excessive value changes are not strongly relevant to a program's behavior, therefore, when constructing a *Runtime State Model*, we only consider a set of key variables.

We select a set of key variables according to two properties,



(a) An Illustration of the *Cumulative Sliding Window* Approach



(b) Properly Merging State Machines with Cumulative Sliding Windows

Figure 5.5: An Example of Merging State Machines with the *Cumulative Sliding Window* Approach

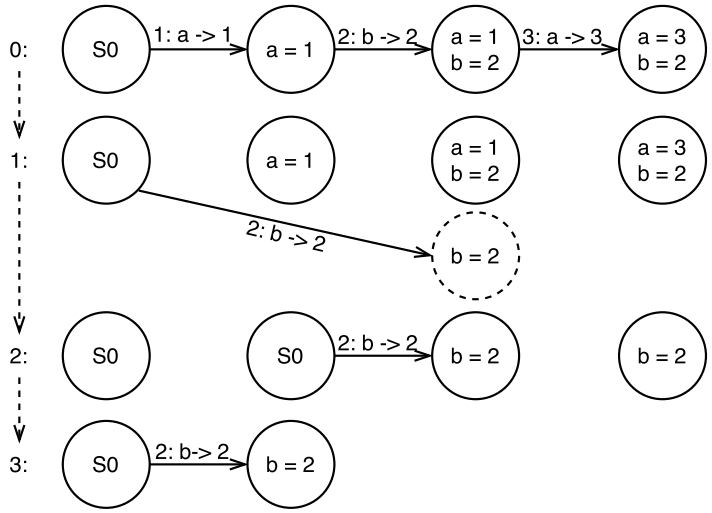


Figure 5.6: An Example of State Machine Pruning

1. The number of changes of a variable v per incoming source src, i.e., $\sum_{i=0}^{n} size(VC_{src,t_i,v})$
2. *Domain* size of a variable v per incoming source, i.e., $size(VAL_{src,v})$, where $VAL_{src,v} = \bigcup_{i=0}^{n} VAL_{src,t_i,v}$

Given that our engine works on a stream of data from multiple sources, i.e., it is not possible to access a whole set of data at any given time, therefore, to enforce the above variable selection scheme, we introduce the *Universal Variable Blacklist*.

The *Universal Variable Blacklist* is essentially a global RDD that contains variables that fail to satisfy our selection predicates. We keep track of the growing variable domains and update the blacklist per windowed input. The *Universal Variable Blacklist* is a superset of failing variables of each incoming source, and is applied to the constructions of state machines for all input.

We incorporate the *Universal Variable Blacklist* in our *Runtime State Generator*, and implement logics to modify state machines in flight without recomputations, which we describe in the next section.

5.3.4 State Machine Pruning

We recognize the necessity to prune a state machine. However, pruning a state machine is not a simple as deriving its subgraph. Consider the example shown in Figure 5.5b, if we

need to leave out variable a, in addition to leaving out the second vertex, and its referencing edges, we also need to change the states described in the third and forth vertex.

We implement logics to leave out a set of variables V in parallel with batch processing constructs provided by Spark GraphX, described as follows,

1. $G.mapTriplet((s_{src}, s_{dst}, (s_{src} \rightarrow s_{dst})) \Rightarrow (s_{src}, s_{dst}, (s_{src} \setminus V \rightarrow s_{dst} \setminus V))$
   We change all edges of the state machine, such that they originate from, and point to new vertices $s' = s \setminus V$, even if the new vertices do not currently exist. This procedure is carried out in parallel with the *mapTriplet* construct of Spark GraphX.

2. $G.mapVertices(s \Rightarrow s \setminus V)$
   We change all vertices of the state machine, such that they represent new states $s' = s \setminus V$. This is also computed in parallel with the *mapVertices* construct.

3. $G' = Graph(G.vertices, G.edges.filter(t \Rightarrow t.s_{src} \neq t.s_{dst}))$
   We create a new graph G' as the new state machine that takes the transformed vertices and edges in step 2. Note that we process the state edges before constructing G' to discard transitions that point back to the same state, and that creating a new graph automatically removes duplicated vertices.

We depict this aforementioned process in Figure 5.6, in which row 0 shows the original state machines, while rows 1 - 3 present the state machines generated by the corresponding steps above. Note that the vertex in dashed line on row 1 indicates that the vertex being pointed to is merely are reference and that the vertex does not currently present in the graph. The vertex is later created in the next step when the vertices are filtered.

5.3.5 Runtime State Validation

The validation of runtime states is the process of generating a *Universal Runtime State Model* given a set of *Runtime State Model*s. Runtime state validation gives a validated state machine $G_v = (S_v, T_v)$ that captures the invariant portion across multiple runtime state machines $G_1 = (S_1, T_1), G_2 = (S_2, T_2), \ldots, G_n = (S_n, T_n)$. A formal definition of constructing a *Universal Runtime State Machine* is described in Section 2.1.5.

Consider the case of input variables changing per execution, e.g., a timestamp, or request id, these variables does not represent the universal state of the program. We argue that the universal states across multiple executions captures a finite set of program states, and thus a more representative summarization of the application's behaviors.

We describe our runtime state validation procedure as follows,

1. Generate a set of state machines $G'_1 = (S'_1, T'_1), G'_2 = (S'_2, T'_2), \ldots G'_n = (S'_n, T'_n)$, by pruning each of the input state machines $G_1 = (S_1, T_1), G_2 = (S_2, T_2), \ldots, G_n = (S_n, T_n)$ to contain only a set of common variables V, such that $\exists v \in s$, $\forall v \in V, \forall s \in S_i, \forall i \in [1, n]$

2. Derive a set of common states S' shared across all of $G'_1 = (S'_1, T'_1), G'_2 = (S'_2, T'_2), \ldots, G'_n = (S'_n, T'_n)$, that $\exists s \in S'_i$, $\forall s \in S', \forall i \in [1, n]$.

3. Derive a set of common transitions T', that $t.src \in S'$ and $t.dst \in S', \forall t \in T'$.

4. Generate a validated graph $G_v = (S_v, T_v)$ that excludes all empty states from $G' = (S', V')$ that $S_v = S' \setminus \{s | size(s) = 0, \forall s \in S'\}$.

The validated runtime state is used in state comparisons, instead of the state machines of each individual input source.

5.3.6 Runtime State Comparison

We analyze the deviation of runtime states by comparing the validated state machines of respective healthy and faulty executions. The comparison of state machines follows the same procedures of the runtime state validation process, except that runtime state comparisons are carried out between two state machines, instead of many. A deviation of state machine $G_1 = (S_1, T_1)$ from state machine $G_2 = (S_2, T_2)$ is defined as state machine $G_d = (S_d, T_d)$, such that $S_d = \{s | s \notin S_2, \forall s \in S_1\}$, and that $T_d = \{t | t.src \in S_d, t.dst \in S_d, \forall t \in T_d\}$.

For some scenarios, simply calculating the differences between two given state machines would give larger than expected search space in finding runtime anomalies. To make this process more effective, we use the χ^2 test to limit our search space of variables that show significant deviations the other state.

To apply the χ^2 significance test, we first calculate the number of appearances of (variable, value) pairs (v, val), across all vertices in the given state machine $G = (S, T)$. The *Frequency of Appearance* (FoA), i.e., $f = (v, val) \rightarrow freq_{(v=val)}$, is then computed for each of the (variable, value) pairs, where $freq_{(v=val)} = appearance(v, val)/appearance(v)$, $\forall (v, val) \in S$. We define the appearance of a variable, $appearance(v)$, as the number of its appearances observed across a state machine.

Assuming that incoming state machines are labeled as *Expected* and *Observed*, where an *Expected* machines is the ground truth states that the *Observed* machine is compared

to. For the case of a χ^2 test, we take the FoA of each variable, $F_{E_v} = f_{e_0}, f_{e_1}, \ldots, f_{e_n}$, from the *Expected* machine, and calculate the FoA of that variable for the *Observed* machine as $F_{O_v} = f_{o0}, f_{o1}, \ldots, f_{0_n}$, where, $\forall i \in [0, n]$,

$$f_{o_i} = \begin{cases} ((v, val_i) \rightarrow freq_{(v=val_i)}) & , \text{if}(v, val_i) \in F_{O_v} \\ ((v, val_i) \rightarrow 0) & , \text{if}(v, val_i) \notin F_{O_v} \end{cases} \quad , \forall (v, val_i) \rightarrow freq_{(v=val_i)} \in F_{E_v}$$

The χ^2 test is conducted on two vectors F_{E_v} and F_{O_v}, for each variable v.

We use the *p-value* of the χ^2 fitness test as the condition to restrict our search space, where, the *p-value* indicates the level of significance whether a hypothesis should be rejected, which we define as, are the *Observed* variable values relevant to the *Expected* variable values in terms of statistical distribution of their *frequency of appearances*. A low *p-value* implies weak relevance.

We refer to the list of variables that failed the hypothesis test as *Anomaly Inducing Variables*, in later section.

Chapter 6

Evaluations

6.1 Experiment Setup

6.1.1 Infrastructure Environment

We conduct our experiments on Shoshin Lab's Styx cluster. The Styx cluster is consisted of 6 high-performance machines. Each styx machine is equipped with dual 2.40GHz Intel Xeon E5-2620 CPUs, supporting a total of 24 hyperthreads per server, as well as 64 GB of main memory and a 1Gbps NIC. All servers run Ubuntu Server 14.04.3 LTS 64-bit with kernel version 3.13.0-65-generic.

For performance isolations, we set up virtual machines across the 6 styx machines, with VirtualBox 4.3.36. Each of the virtual machines is configured to use 8 virtual cores, 8 GB of memory, a 1Gbps NIC, and runs Ubuntu 14.04.3 LTS 64-bit with kernel version 3.19.0-25-generic. These virtual machines are connected through a software bridging interface on each host machine, to the physical switch, and are reachable from each other.

6.1.2 Software & Configurations

We setup RUBiS and Apache Spark on the aforementioned Virtual Machines as follows,

We setup RUBiS instances with a shared database instance, using MySQL (Distribution 5.5.46, Version 14.14). Each RUBiS web service is setup on a separate Virtual Machines with tomcat7 as the web container. Each tomcat7 instance is pre-loaded with our Xtract JVMTI agent specified by the *JAVA_OPTS* environment variable.

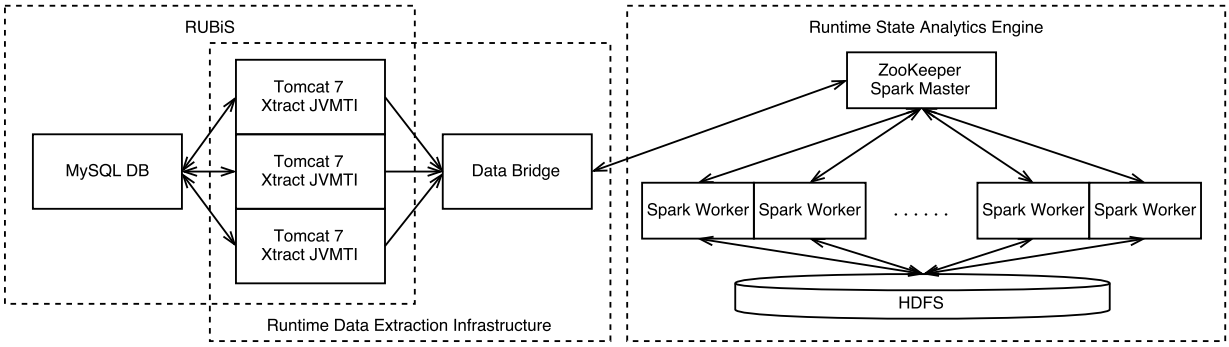


Figure 6.1: Experiment Setup

The *Runtime State Modeling Engine* runs on a standalone Spark Cluster. The Spark cluster uses 9 virtual machines, one for the dedicated Spark master, and one Spark worker on each of the rest. Note that, we increase the amount of memory to 16 GB for each of the Spark worker machines. We run ZooKeeper 3.4.8 in single node mode, on the same machine where the Spark master runs.

The Spark that we use is a customized version based on Spark 1.6, where two modifications are made,

- To provide better support of Protocol Buffer 3.0 that are used extensively in our implementations, we update the dependency of Spark 1.6 to include the Protocol Buffer 3.0 library. To accommodate the serializations of Protocol Buffer message, we implement a customized Kryo serializer for serializations between Spark nodes, and to the external storage system. We describe our modifications in Appendix B.
- To fix the bugs that prevent us from properly checkpointing a graph and running Pregel on GraphX, we applied two proposed patches that are still under review [18, 19].

Given the sub par performance of virtualized I/O, we set up HDFS 2.6 across the 6 physical machines, with one of them as the HDFS master node, and 5 HDFS slaves on each of the rest. HDFS storage is set up on a dedicated spinning hard drive of each node.

6.2 RUBiS

We use RUBiS [15] to evaluate our systems and approaches.

RUBiS is a benchmarking tool for relational databases. It emulates an online bidding system that supports the browsing, searching, selling, bidding, and purchasing of items. There are multiple implementations of RUBiS, however, we use the servlet implementation for simplicity.

RUBiS servlets are basic Java Servlets that are invoked when a corresponding HTTP request is received by the web server. Inside of each RUBiS servlet are logics to parse the request and issue transactions to the underlying database system. If the transaction fails, it will rollback the operation before returning an error back to the caller. Concurrency control of multiple incoming requests is achieved through transactions of the underlying database systems, therefore, no synchronizations are implemented in the RUBiS code.

The RUBiS package includes a client emulator that emulates a client's behavior to browse, search, sell, bid, and purchase an item, based on a probabilistic transition model. The user is able to specify the workload by providing a transition table consisting of the probability of transitions between each pair of operations. The user is also able to specify the number of users, items, and the running time of an emulation. The emulator can generate a list of performance metrics indicating the throughput and latency of each type of requests being served during the emulation. We use the emulator to drive the RUBiS services and for performance evaluations.

In our experiment, we use MySQL as the backend database, and Tomcat7 as the web server container.

6.2.1 Workload

The client emulator of RUBiS emulates the clients' behaviors through its load generator. The load generator takes a state transition table T as input, where each cell of the table $T(s, s')$ is the probability of transition from s to s'. Each state s of T is a simulated user action, i.e., a "click" to execute a particular RUBiS operation. We provide a list of RUBiS operations in Table 6.1. RUBiS operations fall roughly into three categories,

- operations that request **Static Content**s directly served from the web server without interactions with the database. For example, the *Home* operation simply retrieves the *index.html* page from the web server.
- operations that are **DB Read-only** are implemented with one or multiple SQL *SELECT* statements. For example, the *View Item* operation accesses the database to get the details of an item, and is implemented with two separate *SELECT* statements to retrieve information from the *items* and *users* tables.

Category	Operation	Description
Static Content	Home	Go back to the home page
DB Read-only	Browse Categories	Browse the list of categories
	Browse Regions	Browse the list of regions
	Search Items by Categories	Search an item in a provided category
	Search Items by Region	Search an item in a provided region
	View Bid History	View the bidding history of an item
	View Item	View the details of an item
	View User Info	View the details of a user
	About Me	View the details of a logged in user
	Auth	User login authentication
DB Transactions	Register Item	Register a new item
	Register User	Register a new user
	Buy Now	Buy an item with the posted price
	Bid	Bid an item with a user-defined price

Table 6.1: A Summary of RUBiS Operations

- **DB Transactions** are used in operations with output. For example, The *Bid* operations uses database transactions to post the user-defined biding price to the database, and for concurrency control in the case of multiple users bidding the same item. A rollback is applied in the presence of failures to avoid inconsistencies.

The RUBiS client emulator uses one thread per user session simulation. Each session simulates user actions starting from an initial action, and calculates the next action according to the state transition table. In between user actions, a think time is applied using the TPC-W approach [52], i.e., a random distribution with an average of 7 seconds and a maximum of 70 seconds. A session is terminated whenever arriving at the *SessionEnd* state, or when the number of transitions has reached the maximum, whichever comes first, and a new session will be started.

Two types of workloads are used in our evaluations.

- The *Browse-only* workload involves 5% of *Static Content* serving, 75% of *DB Read-only* operations with a *SessionEnd* probability of 20%.
- The *Mixed* workload simulates a more realistic case for Internet bidding services, consisted of 6% of operations involving *DB Transactions*, 1% of *Static Content* serving, and 73% of *DB Read-only* operations with a *SessionEnd* probability of 20%.

Each RUBiS client emulation is consisted of three phases, the *up-ramp* phase, the *evaluation* phase, and the *down-ramp* phase, each associated with different *slow down factor*s. A slow down factor is used in the computation of think time, where $thinkTime = TPCThinkTime() * slowDownFactor$. Performance metrics of the three phases are calculated separately.

6.3 Performance Impact

We use the RUBiS client emulator to evaluate the performance impact of Xtract. Experiments are conducted on two individual setups of the RUBiS service on separate virtual machines, one with the Xtract agent loaded and events enabled, the other without. Both instances share the same dedicated database instance, and run in tomcat7.

We configure the RUBiS client emulator to have a *up-ramp* phase of 5 minutes, a *evaluation* phase of 20 minutes, and a *down-ramp* phase of 1 minute, each associated with a respective *slow down factor* of 2, 1, and 3. The client emulator runs 100 simulated clients in parallel, which produces a workload far from saturating the systems. However, for evaluating the performance overhead of the *Xtract agent*, we argue that more parallel connections will not necessarily reduce the latency for either servers, or make the performance difference more significant.

We run two sets of experiments, one for each of the two workloads, and 5 times for each set. The average throughput and latency during the evaluation phase, as indicated by the RUBiS client emulator is used as the performance metrics.

According to the comparisons in Table 6.2, we see a maximum of 1228x increase in terms of per-request latency and 9x less throughput on RUBiS, when the Xtract agent is loaded and enabled.

6.3.1 Discussions

It is not surprising that performance evaluations on the *Xtract agent* show a maximum of 1228x slow down in terms of latency, given that *Xtract agent* does not, at this point, employ any performance optimizing mechanisms, while the focus of this work is to establish a solution framework, and show the effectiveness of using runtime state models in the context of software runtime anomaly detection.

For discussion, we compare and constrast the performance overhead of related approaches to Xtract.

	avg. throughput (req / s)	avg. latency (ms)
RUBiS without Xtract	15.6	25.8
RUBiS with Xtract	2.4	27411
Slow down factor	6.5	1062.4

(a) With the *Mixed* Workload

	avg. throughput (req / s)	avg. latency (ms)
RUBiS without Xtract	18	29
RUBiS with Xtract	2	35639
Slow down factor	9	1228.9

(b) With the *Browse-only* Workload

Table 6.2: An Evaluation of the Performance Overhead of Xtract

For the case of diagnosing performance issues, X-ray [3] achieves an average runtime overhead of only 2.3% by recording the timestamps at the entry and exit of system calls through binary instrumentation. Magpie [4] has a performance overhead of approximately 4% by logging OS kernel events. However, they are in contrast to our approach to capture variables' value changes that are essential to the construction of *Runtime State Model*s.

ClearView [46] uses Daikon, an x86 binary instrumentation infrastructure, to extract runtime traces. It slows down the execution of the underlying application by a factor of 300, with two major optimizations,

1. It distributes the monitoring of procedures among multiple executions on different machines, where each machine only traces the execution of some selected procedures.

2. It selects a set of variables to monitor according to its control flow graph.

We argue that the performance overhead of the *Xtract agent* is correlated with the number of breakpoints and watchpoints set on various bytecode instructions and class fields. For example, in our experiment, a total of 984 breakpoints are set on 475 respective local variables to extract variable change events, however, only 57 variables are selected with our variable selection scheme. A simple optimization to disable breakpoints set on bytecode instructions that store values to variables excluded by our selection scheme is estimated to reduce the overhead of our approach by a factor of 8.3 (by removing the 418 excessive variables from a total of 475 variables, assuming each variable corresponds to the same number of breakpoints).

We discuss the options to reduce the performance overhead of *Xtract agent* in Section 4.3.

6.4 Anomaly Detection with Runtime State Models

In this section, we evaluate the effectiveness of using the runtime state models in detecting runtime anomalies.

Anomaly detection with runtime models is achieved through the comparison of state machines generated with healthyloads and faultloads, where a healthyload is an execution of a healthy RUBiS service, while a faultload is induced with manually injected faults. We refer to runtime state machines generated from healthyloads as healthy states, and those from faultloads, anomalous states.

6.4.1 Healthy States Generation

We generate healthy runtime states from two simultaneous RUBiS services, without injected faults. They are set up on two separate virtual machines, both using tomcat7 as the web container, and share the same database instance that is deployed on a third virtual machine. Two RUBiS emulators are used to drive the RUBiS instances separately with the *Mixed* workload as described in Section 6.2.1.

We run the emulator for 30 minutes without the *up ramp* and *down ramp* phase with Xtract configured on each of the servers to stream events to our runtime analytics engine simultaneously. The engine constructs *Runtime State Model*s separately and eventually derives a single validated *Universal Runtime State Model*, summarizing the universal behaviors of the healthyload.

6.4.2 Anomalous States Generation

We use fault injection to induce software failures. Despite being used extensively as the target application in relevant research in fault detection, we were unable to find tools or frameworks that automate the injection of faults to an existing J2EE application, which in our case, RUBiS. Instead, we surveyed faults injected by other researchers in their experiments [10, 33, 42, 48], and arrived at the following 4 categories.

- Faults that are external to the application, including, the failures of communications, authentications, etc.
- Accidental references of null pointers, causing *NullPointerException*s
- Unhandled exceptions causing the ungraceful terminations of function calls.
- Concurrency bugs, e.g., deadlocks, starvations, etc.

It is recognized that many of those injections are not applicable to our case, e.g., we used the simplified RUBiS Servlets implementation, and therefore injecting faults to the EJB components is not an option; RUBiS is a multi threaded web applications in that each user session is handled by a separate thread, managed by the underlying tomcat container; The concurrency control of RUBiS Servlets is achieved through database transactions, and therefore, injecting concurrency bugs is also out of the question.

Our runtime state models is a summarization of the program's execution states, therefore, we focus on the faults that could be correlated back to the source code for the sake of the evaluation. That said, we injected a total of 3 types of faults to RUBiS in our experiment, that we describe as follows,

1. **Database failures**
   We introduce intermittent database failures by periodically taking down the MySQL daemon. We refer to this fault as *dbdown*
2. **NullPointerException**s
   We introduce a null pointer references in the RUBiS code by modifying the *init* method in *RubisHttpServlet* class that initializes the database manager. We set the database manager as null. The *RubisHttpServer* class is the base class of all servlet implementations. The code change is shown in Figure 6.3. We refer to the faults as *nulldb* in later sections.
3. **Configuration errors**
   We introduce a configuration error in the RUBiS server by changing the value of *J2eeContainerPath* variable in *Config.java*. The code change is shown in Figure 6.2b. We refer to this fault as *conf* in later sections.

We run the RUBiS client emulator on each of these fault injected servers, collect the runtime data and build one runtime state model per case. These models are compared to the healthy state models to detect software anomalies, which we now describe.

```java
// RubisHttpServlet.java
public abstract class RubisHttpServlet extends BaseRubisHttpServlet {
  private DatabaseConnectionManager _dbMgr = null;

  @Override
  public void init() {
    ...
    this._dbMgr = new DatabaseManager();
    this._dbMgr.init();

    // Fault injected here vvvv
    this._dbMgr = null;
  }

  protected Connection getConnection() {
    return this._dbMgr.getConnection();
  }
}
```

(a) The Injection of *nulldb* Fault

```java
// Config.java
public class Config {
  ...
  // vvvv remove this
  // private static final String J2eeContainerPath = "/var/lib/tomcat7";

  // vvvv change to this
  private static final String J2eeContainerPath = "/var/lib/tomcat6'';
  ...
}
```

(b) The Injection of *conf* Fault

Figure 6.2: Faults Injected to the RUBiS Server

Class Name	method Name	variable name	variable values	pvalue
BrowseRegions	closeConnection	conn	null (100%)	0.32
		stmt	null (100%)	0.32
	regionList	conn	null (100%)	0.02
		rs	null (100%)	0.03
		stmt	null (100%)	0.05
RegisterUser	doGet	conn	null (100%)	0.09
		stmt	null (100%)	0.12

Table 6.3: Anomaly Inducing Variables of the *nulldb* Case

6.4.3 Anomaly Detection with Runtime State Models

In this section, we discuss each case of the aforementioned injected faults, anomalies presented in the runtime state models, and analysis through comparisons to the healthy state to help locate the fault in the source code.

The Case of *nulldb*

The *nulldb* fault, as shown in Figure 6.3, induces a null *_dbMgr* used by *all* RUBiS operations to communicate with the database. Through comparisons of its fault state, to the healthy state, we arrived at the following *Anomaly Inducing Variables* that deviate from the regular state in Table 6.3. It shows that the variables *conn, stmt* are 100% *null* in *BrowseRegions* and *RegisterUser*, indicating strong deviations from what are obverved in the healthy states.

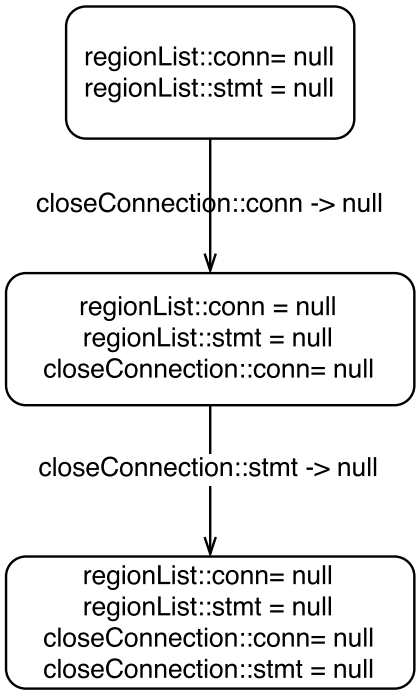
Take the *BrowseRegion* operation as an example. From the RUBiS server's *BrowseRegion* implementation (summarized in Figure 6.3a), it is not hard to determine that *conn* and *stmt* passed to the *closedConnection* method should not be *null* under regular circumstances.

To further diagnose the actual fault in the system, we inspect the faulty state transitions, given in Figure 6.3b. A state with *conn* and *stmt* being null could be at the beginning of the method *regionList*, however, a direct transition from this state that *closeConnection::conn* $\rightarrow$ *null* and *closeConnection::stmt* $\rightarrow$ *null* indicates that those variables are never properly initialized, and that an exception is probably thrown that calls the *closeConnection* method with null parameters.

Intuitively, one would then inspect the *getConnection()* function that eventually leads to the fault that we injected.

```
1  // BrowseRegions.java
2  class BrowseRegions extends
       RubisHttpServlet {
3    ...
4    // conn, stmt = null;
5    private void regionList(...) {
6      try {
7        conn = this.getConnection();
8        stmt = conn.prepareStatement
             (...);
9        ...
10       closeConnection(conn, stmt);
11     } catch (...) {
12       closeConnection(conn, stmt);
13     }
14   }
15
16   private closeConnection(...) {
17     ...
18     conn.close()
19   }
20 }
```

(a) Code Snippet of *BrowseRegion* to Execute Queries



(b) Anomalous State Transitions of the *nulldb* Case

Figure 6.3: Anomaly Detection of the *nulldb* Case

Class Name	method Name	variable name	variable values	pvalue
ViewBidHistory	doPost	connAlive	false (29%)	0.30
StoreCommit	doPost	conn	null (24%)	0.22
		stmt	null (49%)	0.41

Table 6.4: Anomaly Inducing Variables of the *dbdown* Case

Class Name	method Name	variable name	variable values	pvalue
ServletPrinter	printFile	fis	null (100%)	0.019

Table 6.5: Anomaly Inducing Variables of the *conf* Case

The Case of *dbdown*

Unlike the case of *nulldb*, *dbdown* suffers from intermittent database failures induced by our injected fault that periodically takes down the database service, i.e., only a portion of requests to the RUBiS server would fail due to the database failure.

It is observed that the *conn* and *stmt* variables take a *null* value for 30% of the time instead of 100 as in the *nulldb* case, however, are significant enough deviations as determined by the χ^2 test. It is also observed that the possibility of the *connAlive* variable taking a *false* value is also higher, with a probability of 29% compared to 7% in healthy states. We summarize these Anomaly Inducing Variables of this case in Table 6.4.

Given that this is an external failure that is irrelevant to the RUBiS implementation, our runtime states comparison didn't show anomalous state transitions as in the other cases.

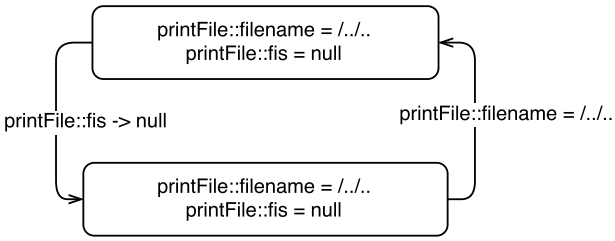
The Case of *conf*

We introduce a configuration error to the RUBiS server, by modifying the path where RUBiS uses to locate its HTML files. The file path is defined in the *Config.java* file, using a variable named *J2eeContainerPath*. The value of *J2eeContainerPath* is supposed to be pointing to the location where the tomcat container is install, which in our case, */var/lib/tomcat7*.

We inject a fault by changing this value to */var/lib/tomcat6* so that RUBiS would fail to access its HTML files. We observe that this configuration is only used by the *printFile* function of the *ServletPrinter* class. We summarize this code change in Figure 6.2b.

```
1  public class ServletPrinter {
2    void printFile(String filename) {
3      FileReader fis = null;
4      try
5      {
6        fis = new FileReader(filename);
7        ...
8      } catch (...) { ... }
9    }
10 }
```

(a) Logics of *filename* Being Used in *printFile*



(b) Anomalous State Transitions of *fis* and *filename*

Figure 6.4: Anomaly Detection of the *conf* Case

Comparing the fault states and the healthy states gives two *Anomaly Inducing Variables*, *ServletPrinter::printFile::fis* and *ServletPrinter::printFile::filename*. We observe a χ^2 p-value of 0.02 for *fis* being *null* 100% of the time in the fault states, while this number is only 16% in the healthy states. For *filename*, since none of its values match the ones in the healthy state, *NaN* is given.

Looking at the runtime states involving both variables, a state transition depicted in Figure 6.4b is obvserved. According to the logic of *printFile*, as shown in Figure 6.4a, *fis* is first assigned as *null* before being initialized to a new *FileReader* object. The *FileReader* object takes *filename* as the path to file to be read. However, as revealed in our state model, *fis* is never initialized, therefore, one would conclude that line 3 is never successfully executed with a high possibility of throwing an exception and jumping to the *catch* block.

By comparing the values of *filename* in both states, it is observed that the value has been changed to */var/lib/tomcat6* from */var/lib/tomcat7*, and the fault is therefore located.

Chapter 7

Conclusions and Future Work

In this work, we introduce the *Runtime State Model* in the context of software anomaly detection and fault localization to achieve the automated detection of generic faults, at a source code granularity.

To automate the extraction of runtime traces and the construction of *Runtime State Model*s, we designed and implemented *Xtract*, a facility that automatically extracts runtime traces from the Java Virtual Machines and constructs *Runtime State Model*s for multiple simultaneous Java applications. This facility consists of two separate entities,

- the *Runtime Data Extraction Infrastructure* retrieves runtime traces of the applications, without prior knowledge of the target applications. It makes use of the JVMTI interfaces to extract variables' value change events directly from the Java Virtual Machine, using Breakpoints and Event Notification constructs.
- the *Runtime State Analytics Engine* is a scalable and massively parallel *Runtime State Model* construction and analytics engine that works on multiple simultaneous runtime event streams. It is implemented entirely on top of Apache Spark to achieve efficient batch and graph processing.

To construct *Runtime State Model*s from real-time runtime event streams, we adopt temporal joins, multi-thread correlation and variable selection schemes to tackle the challenges of broken windows and state explosion. To compare and analyze the runtime states, we introduce the notion of *Anomaly Inducing Variables* with χ^2 significance test on variables' *frequency of appearance* to limit our search space.

We evaluate the effectiveness of applying *Runtime State Model*s in the detection of software anomalies by injecting three types of faults to the RUBiS service, including null pointer referencing, external database failures, and a configuration error. Our evaluation shows evidence that the facility and approaches might be effective in detecting runtime anomalies and providing useful information to help locate the faults, for all three cases.

Although *Xtract* works on Java applications, through extracting runtime traces directly from the Java Virtual Machines, we argue that it is purely a decision to simplify our implementations, since Java provides mature tool sets to access runtime information. The *Runtime State Model*s do not make any assumptions of the target applications, and could also be applied to the diagnosis of applications in compiled native binaries with proper binary instrumentation techniques.

7.1 Discussions and Future Work

Despite the fact that we have achieved our initial goal by providing a working solution framework that automates the process of constructing and analyzing *Runtime State Model*s, and showed evidence that it could be effective to use *Runtime State Model*s in the detection of anomalies and providing useful information to help locate the faults. We discover the following in need of further efforts that we list as future work.

Xtract's most significant limitation is performance. Our current implementation does not employ any optimizations, making it 1000x slower when it is enabled. We discover that the overhead is vastly introduced by the breakpoints set to extract runtime traces which pause the execution of threads and disable code optimizations. We argue that this could be optimized through the use of a feedback loop that propagates a list of insignificant variables, e.g., those not captured in the validated models, back to *Xtract*, so that those breakpoints could be disabled to reduce the runtime overhead. For our evaluation on RUBiS, it is estimated that disabling excessive breakpoints could reduce the overhead by a factor of 8.3. It is also observed that code analysis techniques could be helpful in this case to determine insignificant variables.

Reducing the performance overhead of *Xtract* also makes it possible to be used on large-scale and performance critical systems. This work uses RUBiS as the target application, which is a relatively small-scale web application from research communities; however, the lack of its source code commit history forces us to use fault injection for our evaluation. By employing more optimizations to *Xtract*, we can conduct evaluations on larger-scale production ready systems with bug tracking histories, enabling the possibility to evaluate our approach on real software bugs.

We observe that *state noise* in the constructed state models, i.e., variables that are included in the state model, but not significant, reduces the effectiveness of anomalous state identifications. We attribute the *state noise* to our coarse-grained variable selection scheme that only considers the *velocity* and *domain size* of the variables. It is recognized that finer-grained variable selection schemes could be plugged in to our engine, including, the use of pattern mining to select variables that are more likely to appear together, and through code analysis to determine the significance of each variable.

APPENDICES

Appendix A

Core Interfaces and Proto Definitions of Xtract

APIs provided by Xtract are implemented with an RPC service. To help better understand what and how information is retrieved from the JVM, we now describe core interfaces and proto message definitions of Xtract.

A.1 Core Proto Message Definitions

9 proto messages are used extensively throughout our system, and are important to the understanding of the infrastructure, as shown in Figure A.1. We discuss each of those as follows.

- A ***JavaObject*** is a proto message that describes a Java heap object. A *JavaObject* is either a typical class instance or one of the following three special cases, a *JavaString* that describes a string type in the Java heap defining the length and content of the string, a *JavaArray* that describes a primitive array defining the length and content of the array, or a *JavaClass* that describes a class object defining the name, methods info and static fields of the class.
- Each *JavaObject* has reference information defined as ***JavaObjectReferences***, unless they belong to one of those special cases. A *JavaObjectReferences* proto contains reference info of a heap object that includes its fields, static fields, referrers, its class object, etc.

```
message JavaObject {
  message JavaString {
    optional int64 length = 1;
    optional bytes string_content = 2;
  }

  message JavaArray {
    optional int64 length = 1;
    repeated JavaPrimitive array_content = 2;
  }

  message JavaClass {
    optional string class_name = 1;
    repeated JavaMethodInfo java_methods = 2;
    repeated JavaObjectField java_object_fields = 3;
  }

  // next tag: 6
  oneof object_value {
    JavaClass java_class = 1;
    JavaString java_string = 2;
    JavaArray java_array = 3;
  }

  optional JavaObjectMetadata metadata = 4;
  optional JavaObjectReferences references = 5;
}
```

(a) Proto Definitions of *JavaObject*

```
message JavaObjectReferences {
  // next tag: 11
  repeated JavaObjectField fields = 2;
  repeated JavaObjectField static_fields = 3;
  repeated JavaObjectField object_array = 6;
  repeated JavaObjectMetadata referrers = 7;
  optional JavaObjectMetadata class_object = 10;
}
```

(b) Proto Definitions of *JavaObjectReferences*

Figure A.1: A List of Core Proto Message Definitions of Xtract

```
message JavaObjectField {
  // next tag: 10
  optional int32 parent_object = 1;
  optional int64 field_id = 2;
  oneof field_name_or_index {
    string field_name = 3;
    int32 field_index = 4;
  }

  optional string field_class_name = 5;

  oneof field_value {
    JavaObjectMetadata object_field = 6;
    JavaPrimitive primitive_field = 7;
  }

  optional string parent_object_class_name = 9;
}
```

(c) Proto Definitions of *JavaObjectField*

```
message JavaObjectMetadata {
  // object category
  // next tag: 2
  enum ObjectType {
    HEAP_OBJECT = 0;
    CLASS_OBJECT = 1;
    PRIMITIVE_ARRAY = 2;
    STRING = 3;
  }

  required ObjectType object_type = 1;

  // object properties
  // next tag: 14
  optional int64 size = 10;
  optional string class_name = 11;
  optional int32 id = 12;
}
```

(d) Proto Definitions of *JavaObjectMetadata*

Figure A.1: A List of Core Proto Message Definitions of Xtract (cont.)

```
message JavaPrimitive {
  oneof primitive_value {
    bool bool_value = 2;
    double double_value = 3;
    float float_value = 4;
    int32 int_value = 5;
    int64 long_value = 6;
    int32 short_value = 7;
    int32 char_value = 8;
    int32 byte_value = 9;
  }
}
```

(e) Proto Definition of *JavaPrimitive*

```
message JavaMethodInfo {
  optional int64 method_id = 1;
  optional string method_name = 2;
  optional bytes bytecodes = 3;
  optional int32 parent_class_id = 4;
  optional string parent_class_name = 5;
  optional int32 max_slot = 6;
}
```

(f) Proto Definition of *JavaMethodInfo*

```
message SetBreakpointParam {
  optional int64 method_id = 1;
  optional int64 location = 2;
}
```

(g) Proto Definition of *SetBreakpointParam*

```
message JavaLocalVariable {
  optional string name = 1;
  optional string signature = 2;
  oneof value {
    JavaPrimitive primitive_value = 3;
    string string_value = 4;
    bool null_object = 5;
  }
}
```

(h) Proto Definition of *JavaLocalVariable*

Figure A.1: A List of Core Proto Message Definitions of Xtract (cont.)

```
message JvmEventNotification {
  enum EventType {
    FIELD_MOD_EVENT = 0;
    BREAKPOINT_EVENT = 1;
  }

  message JavaFieldModificationEvent {
    optional JavaExecContext context = 1;
    optional JavaObjectField field = 2;
    oneof java_object_or_primitive {
      JavaObject new_object = 3;
      JavaPrimitive new_value = 4;
    }
  }

  message JavaBreakpointEvent {
    optional int64 timestamp = 1;
    optional int64 thread_id = 2;
    optional int64 method_id = 3;
    optional int64 location = 4;
    optional string method_name = 5;
    optional string class_name = 6;
    repeated JavaLocalVariable local_variables = 7;
  }

  required EventType event_type = 1;
  oneof event {
    JavaFieldModificationEvent field_mod_event = 2;
    JavaBreakpointEvent breakpoint_event = 3;
  }

  optional string from_hostname = 4;
}
```

(i) Proto Definition of *JvmEventNotification*

Figure A.1: A List of Core Proto Message Definitions of Xtract (cont.)

- Each object field is represented by the type ***JavaObjectField***. A *JavaObjectField* proto describes the parent object id, parent object class name, field id, field name or array index, field class name, and field value, i.e., the object's metadata if the field points to an object instance, or a primitive if it is a primitive field.

- To make our Java heap representations simple and compact, we divide a Java heap object into a *JavaObject* type that stores the actual values and references, and a ***JavaObjectMetadata*** type that stores the type, the size, the id and the class name of the object. *JavaObjectMetadata* is used instead of *JavaObject* when the identification but the actual value of an object is needed.

- A ***JavaPrimitive*** is a proto message to describe a primitive value in Java. It is defined as one of the values of a primitive type in the Java language, being *boolean*, *char*, *short*, *int*, *long*, *float*, *double* or *byte*.

- A ***JavaMethodInfo*** proto message describes a Java method in the JVM. It records the id, name, the declaring class, bytecodes and max slot number of the method.

- The ***SetBreakpointParam*** proto identifies a unique breakpoint in the JVM, containing information of the method id and instruction location at which a breakpoint needs to be set. It is used as the parameter of the *SetBreakpoint* and *ClearBreakpoint* function.

- A ***JavaLocalVariable*** message describes a Java local variable in the JVM. It contains information of the name, signature, and values of a Java local variable.

- ***JvmEventNotification*** is a universal proto message used to describe a JVMTI event sent from a JVMTI agent. In our current implementation, it supports two types of events, the *FieldModification* event and the *Breakpoint* event, as indicated by the *EventType* enum. It contains one of the two event proto messages, *JavaFieldModificationEvent* and *JavaBreakPointEvent*, each describing the details of a respective event.

These proto messages are foundational to the RPC service, and are used as parameters and return types of our RPC interfaces, which we now describe.

A.2 Core Interfaces

Our RPC interfaces fall into three major categories,

1. The Control Interfaces that configure Xtract on the fly.
2. The Output Interfaces that external entities invoke to extract information from the JVM.
3. The Push Interfaces that send event notifications to the external entity.

An interface is an RPC stub definition that specifies the name, and request / response types of an RPC request. In our case, the RPC interfaces are defined in the *.proto* files in a language-independent syntax (referred to as gRPC syntax), and are later compiled into the server and client code of languages of our choices. For clarity and readability, we present 9 of our core interfaces in gRPC syntax in Figure A.2. Note that each interface takes exactly *one* proto message as the request, and exactly *one* proto message as the response.

We discuss each of the interfaces as follows.

- The ***SetGlobalPolicy*** interface configures various behaviors of Xtract, e.g., whether to use delta snapshotting, whether to enable event notifications from a global perspective. It takes a request of type *XtractGlobalPolicy*.
- The ***ToggleEventNotifications*** interface enables or disables a JVMTI event. It takes a request of type *JavaEventManagement* that defines whether to enable or disable a particular JVMTI event.
- The ***SetBreakpoints*** interface sets a breakpoint at a bytecode location specified in the RPC request. It takes a request of type *MultipleSetBreakpointParams*, a repetition of *SetBreakpointParams* messages defined in Figure A.1g.
- The ***ClearBreakpoints*** interface clears the breakpoints set at various bytecode locations. It takes a request of type *MultipleBreakpointParams*.
- The ***ClearWatchpoints*** interface clears the watchpoints set on various class fields. The request of type *MultipleClassFields* contains class and field id pairs on which the watchpoints need to be cleared.
- The ***GetLoadedClasses*** interface gets a list of classes currently loaded in the JVM. It takes an empty request, and produces a response of type *MultipleClassNames* that contains a list of class name strings identifying those classes.

- The ***GetHeapObjects*** interface gets a stream of *JavaObject* messages each describing a Java heap object, as defined in Figure A.1a, from the JVM. Those objects either are or are reachable from the objects of given classes as indicated in the request. The request *MultipleClassNames* contains a list of class name strings identifying those classes. Note that this interface implements the streaming interface of gRPC, for which, the server and client sends and receives objects in a similar fashion to writing byte streams to a TCP connection as discussed in Section 4.1.1.

- The ***GetClassMethods*** interface gets a list of method information for each class given as the request. It takes *MultipleClassNames* as request containing a list of class name strings identifying loaded classes in the JVM, and produces a response of type *MultipleJavaMethodInfo*, a repetition of *JavaMethodInfo* messages as defined in Figure A.1f.

- The ***PushJvmEvent*** interface sends a stream of JVMTI events from Xtract to external entities. It takes one request of type *JvmEventNotification* as defined in Figure A.1i. It is a universal interface used for all event notifications that are later distinguished by the *EventType* field of the *JvmEventNotification* message. Note that this interface implements the gRPC streaming interface.

```
rpc SetGlobalPolicy(XtractGlobalPolicy)
    returns (XtractStatus) {}
rpc ToggleEventNotifications(JavaEventManagement)
    returns (XtractStatus) {}
rpc SetBreakpoints(MultipleSetBreakpointParams)
    returns (XtractStatus) {}
rpc ClearBreakpoints(MultipleSetBreakpointParams)
    returns (XtractStatus) {}
rpc ClearBreakpoints(MultipleClassFields)
    returns (XtractStatus) {}
```

(a) Definitions of Control Interfaces

```
rpc GetLoadedClasses(XtractEmpty)
    returns (MultipleClassNames) {}
rpc GetHeapObjects(MultipleClassNames)
    returns (stream JavaObjects) {}
rpc GetClassMethods(MultipleClassNames)
    returns (MultipleJavaMethodInfo) {}
```

(b) Definitions of Output Interfaces

```
rpc PushJvmEvent(stream JvmEventNotification)
    returns (XtractStatus) {}
```

(c) Definitions of Push Interfaces

Figure A.2: A List of Core Interfaces of Xtract

Appendix B

Issues of Protocol Buffer Integration with Spark 1.6

Protocol Buffer is used extensively in this Thesis, in both of the Runtime Data Extraction Infrastructure and the Runtime State Modelling Engine as descibed in Chapter 4 and Chapter 5 respectively. However, Protocol Buffer is not natively supported in Spark 1.6 used as a fundamental infrastructure in our Runtime State Modelling Engine.

In this chapter, we describe the approaches taken the work around this problem.

B.1 Enabling Protocol Buffer 3.0 in Spark 1.6

The latest version of Spark, Spark 1.6, uses Protocol Buffer 2.5 as one of its dependencies, however, Protocol Buffer 2.5 lacks fundamental support for modern features that are used extensively in our message definitions, including the support of *oneof* construct, etc. This causes *MethodNotFound* runtime exceptions when the underlying JVM is trying to serialize our proto messages with newer Protocol Buffer implementations.

To solve this problem, we compiled a customized version of Spark 1.6 from the source code, modifying its Protocol Buffer dependency from version 2.5 to the latest version 3.0 beta 2. This is easily achieved by changing the value of *protobuf version* from *2.5.0* to *3.0.0-beta-2* in the *pom.xml* file.

```
<protobuf version>3.0.0-beta-2</protobuf version>
```

B.2 Serializing Protocol Buffer Messages in Spark 1.6

Enabling Protocol Buffer 3.0 Support in Spark 1.6 solves the *MethodNotFound* runtime exceptions, however, the current Spark version failed to serialize RDDs containing proto messages during shuffles, giving *ClassNotFound* exceptions for not being able to find the classes where we define our proto messages, when using Java Serializers, and *UnsupportedOperation* exceptions, when using the Kryo Serializers.

We work around this problem by defining a custom kryo serializer for proto messages. The custom Kryo Serializer for Protocol Buffer messages uses the *toByteArray()* and *parseFrom()* methods provided by Protocol Buffer to serialize and materialize a proto message. To tell Spark which Kryo serializers to use when serializing an object, it is also necessary to provide a user-defined Spark Kryo Registrator to specify or override the Kryo serializers to use for various object types. We list our implementations of the custom Protocol Buffer Kryo Serializer and class registering logics in Figure B.1

B.3 Saving Protocol Buffer RDDs to Object Files

In our Runtime State Modelling Engine, the communications between different Spark applications are achieved by saving and reading object files from the HDFS. These object files are essentially serialized Spark RDDs that could be materialized back into RDD objects in Spark. Saving and reading an RDD to / from an object file could be achieved with the *saveAsObjectFile* / *objectFile* methods defined as members of *RDD* and *SparkContext* respectively.

```
// class RDD[T]
def saveAsObjectFile(path: String): Unit
```

```
// class SparkContext
def objectFile[T](path: String, minPartitions: Int =
    defaultMinPartitions): RDD[T]
```

However, both of these two methods serializes RDDs using the Java Serializer that does not work well with Protocol Buffer, and therefore, with RDDs containing Protocol Buffer messages. As a workaround, we implement helpers to save and read RDDs to / from object files using the Kryo serializers as specified by the Spark Kryo Registrator, which in our case, the one defined in Figure B.1b.

```
public class ProtobufKryoSerializer <M extends GeneratedMessage> extends
    Serializer<M> {
  @Override
  public M read(Kryo arg0, Input arg1, Class<M> arg2) {
    int serialized_message_length = arg1.readInt();
    if (serialized_message_length == 0) {
      return null;
    }

    byte[] serialized_message = arg1.readBytes(serialized_message_length);
    Method parseFromMethod = getParseFromMethod(arg2);

    try {
      return (M) parseFromMethod.invoke(arg2, serialized_message);
    } catch (IllegalAccessException | IllegalArgumentException |
       InvocationTargetException e) {
      e.printStackTrace();
      return null;
    }
  }

  @Override
  public void write(Kryo arg0, Output arg1, M arg2) {
    byte[] serialized_message = arg2.toByteArray();

    arg1.writeInt(serialized_message.length);
    arg1.writeBytes(serialized_message);
  }

  public Method getParseFromMethod(Class<M> c) {
      try {
        return c.getMethod("parseFrom", byte[].class);
      } catch (NoSuchMethodException | SecurityException e) {
        e.printStackTrace();
        return null;
      }
  }
}
```

(a) Definitions of the Custom Kryo Serializer for Protocol Buffer Messages

Figure B.1: Custom Kryo Serializer for Protocol Buffer Messages in Spark 1.6

```
class XtractKryoRegistrator extends KryoRegistrator {
  override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[JavaLocalVariable], new ProtobufKryoSerializer[
        JavaLocalVariable]())
    kryo.register(classOf[JavaPrimitive], new ProtobufKryoSerializer[
        JavaPrimitive]())
    kryo.register(classOf[JavaBreakpointEvent], new ProtobufKryoSerializer[
        JavaBreakpointEvent]())
  }
}

conf.set("spark.kryo.registrator", classOf[XtractKryoRegistrator].getName)
```

(b) Registering Protocol Buffer Message Classes

Figure B.1: Custom Kryo Serializer for Protocol Buffer Messages in Spark 1.6 (cont.)

When serializing an RDD, the helper first serializes the RDD object to a byte stream that is later written to an external file system as a sequence file using the *saveAsSequenceFile* API of Spark that takes care of communications with various file systems, including HDFS. Reading an RDD object from an external file system reverses the aforementioned procedure. We list the implementations of the helper in Figure B.2.

```
object XtractKryoFileSupport {
  def saveAsKryoObjectFile[T: ClassTag](kryoSerializer: KryoSerializer, rdd:
      RDD[T], path: String) {
    rdd.mapPartitions(iter => iter.grouped(10).map(_.toArray))
      .map(x => (NullWritable.get(), new BytesWritable(kryoSerialize(
          kryoSerializer, x))))
      .saveAsSequenceFile(path)
  }

  def fromKryoObjectFile[T: ClassTag](sc: SparkContext, minPartitions: Int,
     path: String): RDD[T] = {
    val kryoSerializer = new KryoSerializer(sc.getConf)
    sc.sequenceFile(path, classOf[NullWritable], classOf[BytesWritable],
        minPartitions)
      .flatMap(x => kryoDeserialize[Array[T]](kryoSerializer, x._2.getBytes)
          )
  }

  def kryoSerialize[T: ClassTag](kryoSerializer: KryoSerializer, o: T):
     Array[Byte] = {
    val kryo = kryoSerializer.newKryo()
    val bos = new ByteArrayOutputStream()
    val out = new Output(bos)
    kryo.writeClassAndObject(out, o)
    out.close()

    bos.toByteArray
  }

  def kryoDeserialize[T: ClassTag](kryoSerializer: KryoSerializer, bytes:
     Array[Byte]): T = {
    val kryo = kryoSerializer.newKryo()
    val bis = new ByteArrayInputStream(bytes)
    val in = new Input(bis)
    val obj = kryo.readClassAndObject(in)

    obj.asInstanceOf[T]
  }
}
```

Figure B.2: The Implementation of to Save and Load RDD Objects from Object Files

References

[1] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*. IEEE Computer Society Press, 1976.

[2] Jennifer M Anderson, Lance M Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R Henzinger, Shun-Tak A Leung, Richard L Sites, Mark T Vandevoorde, Carl A Waldspurger, and William E Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 1997.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012.

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

[5] Peter C Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, 1995.

[6] Peter Lawrey Ben Evans. Introduction to JIT compilation in Java HotSpot VM. http://www.oraclejavamagazine-digital.com/javamagazine_open/20120506?pg=49#pg49, May 2012.

[7] Sapan Bhatia, Abhishek Kumar, Marc E Fiuczynski, and Larry L Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2008.

[8] George Candea, James Cutler, and Armando Fox. Improving availability with recursive microreboots: a soft-state system case study. *Performance Evaluation*, 56(1):213–248, 2004.

[9] George Candea and Armando Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[10] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(2):175–190, 2006.

[11] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, 2004.

[12] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 1999.

[13] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[14] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2002.

[15] OW2 Consortium. RUBiS: Rice University Bidding System. http://rubis.ow2.org/.

[16] Oracle Corporation. JVM Tool Interface (ver 1.2). https://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html.

[17] Barb Darrow. Amazon explains database glitch that impacted big customers. http://fortune.com/2015/09/23/amazon-database-glitch/, September 2015.

[18] Apache Spark Developers. SPARK-10335: graphx connected components fail with large number of iterations. https://issues.apache.org/jira/browse/SPARK-10335.

[19] Apache Spark Developers. SPARK-14804: graph vertexrdd/edgerdd checkpoint results classcastexception. https://issues.apache.org/jira/browse/SPARK-14804.

[20] OpenJDK Developers. OpenJDK 7 HotSpot VM Source Repository. http://hg.openjdk.java.net/jdk7/jdk7/file/ee67ee3bd597.

[21] Sun Developers. The Java Hotspot performance engine architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html, 2007.

[22] Stephen Elliot. Devops and the cost of downtime: Fortune 1000 best practice metrics quantified. *International Data Corporation (IDC)*, 2014.

[23] Ben Evans. Understanding the Java HotSpot VM code cache. http://www.oraclejavamagazine-digital.com/javamagazine_open/20130708?pg=42#pg42, July 2013.

[24] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2007.

[25] OpenJDK Foundation. JDK-4228507: Support Local Variable and Array Element Watchpoints. https://bugs.openjdk.java.net/browse/JDK-4228507.

[26] Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*. IEEE, 1998.

[27] Saeed Ghanbari, Ali B. Hashemi, and Cristiana Amza. Stage-aware anomaly detection through tracking log points. In *Proceedings of the 15th International Middleware Conference*. ACM, 2014.

[28] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014.

[29] Dan Goodin. Boeing 787 Dreamliners contain a potentially catastrophic software bug. http://arstechnica.com/information-technology/2015/05/boeing-787-dreamliners-contain-a-potentially-catastrophic-software-bug/, August 2015.

[30] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of The Winter 1992 USENIX Conference*. Citeseer, 1991.

[31] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*. IEEE, 1995.

[32] Amazon Inc. Amazon EC2 Service Level Agreement. https://aws.amazon.com/ec2/sla/, June 2013.

[33] Miao Jiang. *Modeling Management Metrics for Monitoring Software Systems*. PhD thesis, University of Waterloo, 2011.

[34] David B Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the 7th Symposium on Principles of Distributed Computing (PODC)*. ACM, 1988.

[35] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 1987.

[36] Leslie Lamport. Computation and state machines, 2008.

[37] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 2001.

[38] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science (vol. B)*, pages 1157–1199. MIT Press, 1991.

[39] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Pearson Education, 2014.

[40] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*. ACM, 2010.

[41] Mohammad A Munawar, Miao Jiang, Thomas Reidemeister, and Paul AS Ward. Filtering system metrics for minimal correlation-based self-monitoring. In *Proceedings of the 3rd International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 2009.

[42] Mohammad Ahmad Munawar. *Adaptive Monitoring of Complex Software Systems using Management Metrics*. PhD thesis, University of Waterloo, 2009.

[43] Mohammad Ahmad Munawar, Michael Jiang, and Paul AS Ward. Monitoring multi-tier clustered systems with invariant metric relationships. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-managing Systems*. ACM, 2008.

[44] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.

[45] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical report, Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[46] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, 2009.

[47] Sharon E Perl and William E Weihl. Performance assertion checking. In *ACM SIGOPS Operating Systems Review*. ACM, 1994.

[48] Thomas Reidemeister. *Fault Diagnosis in Enterprise Software Systems Using Discrete Monitoring Data*. PhD thesis, University of Waterloo, 2012.

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

[50] 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)*. USENIX Association, 2011.

[51] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale

distributed systems tracing infrastructure. Technical report, Technical report, Google, 2010.

[52] Wayne D Smith. TPC-W: benchmarking an e-commerce solution. Transaction Processing Council, 2002.

[53] Apache Spark. Apache spark - lightening fast cluster computing. spark.apache.org.

[54] Christine Harper Stephanie Ruhle and Nina Mehta. Knight trading loss said to be linked to dormant softwaret. http://www.bloomberg.com/news/articles/2012-08-14/knight-software, August 2012.

[55] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SIGOPS)*. ACM, 2009.

[56] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2010.

[57] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems*, 2012.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.

[59] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[60] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014.