Deep Root-Cause Performance Analysis: Integration and Navigation from a Load Testing Tool

Yoonki Song Tao Xie
Department of Computer Science
North Carolina State University, Raleigh, USA
{ysong2, txie}@ncsu.edu

ABSTRACT

Software performance testing helps testers to validate and verify software quality in terms of response time, reliability, and scalability. Many performance-testing tools can measure client-side response time and correlated it with collected application transaction monitoring data. Such correlation can break down client-side response time into its constituent parts on the various tiers of the server(s) under test. To avoid high overhead, some other performance tools for Java applications perform runtime profiling by using JVM profiling with statistical sampling. However, such an imprecise profiling mechanism poses challenges for performance analysis. It is challenging to identify the correlation between a transaction or time-range on the client side and the profiling information at a server's JVM. Therefore, it is difficult to provide user-friendly drill-down navigation from client-side response time or time-range to determine problem spots and bottlenecks in the JVM(s) on the server sides because of a lack of correlation between the transactions or time-range driven by a load testing tool at the client side and the runtime sample data collected by the JVM profiling tools at the server sides. In the paper, we have developed a new technique and tool to help to address the preceding issues in deep root-cause performance analysis. We evaluate the accuracy, efficiency, and effectiveness of our approach by conducting experiments with several real-world web applications.

1. INTRODUCTION

Performance testing and analysis tools help testers to identify and diagnose the presence and cause of application or system bottlenecks. In performance testing, testers create loads to expose problems and observe failures. In the analysis and diagnosis against the testing results, they try to get the location of the root causes of the observed failures. Usually, deep root-cause performance analysis requires correlation of test results from the client side and traces/events information from the server side.

Many performance-testing tools [7] [6] can measure client-side response time and correlated it with collected application transaction monitoring data from the server side. Application Response Measurement (ARM) [1] used by the tools provides a common way to get profiling information and allows to measure application availability, application performance, and end-to-end transaction response time.

For non ARM-instrumented servers running Java applications, Java Virtual Machine Tool Interface (JVMTI) [4] feature of Eclipse Test and Performance Tools Platform (TPTP) [9] allows performance-testing tool to collect full traces from the non ARM-instrumented servers. One problem is here is that the collection of full traces is expensive and slow. To reduce the cost of the collection of full traces from the servers, a selective instrumentation should be conducted. However, such an imprecise profiling mechanism poses challenges for performance analysis because it is difficult to identify the correlation between a transaction (or time-range) on the client and the profiling information at a server's JVM.

In our approach, we show that a technique to recover information that is available in full traces and necessary for performance analysis from sampled traces. With the sampled traces, we show a robust technique for matching the sampled method calls with the start and end points identified

We have developed a new tool, an Eclipse plug-in, for reading in performance tests and traces in the XML format and identifying the start and end point of a transaction or a request among the method execution on the server side. The tool helps testers to perform deep root-cause analysis by integrating with a load-testing tool and navigating the performance results.

In the evaluation, we compared the precision and recall of the set of we compared the runtime cost of the full traces with the cost of the sampled traces. The result is.

Contributions of our work are the followings:

- We provide an approach for supporting performance analysis in the absence of ARM.
- We suggest a technique for supporting performance analysis with sampled traces rather than fully collected traces.

The remainder of this paper is organized as follows. Section 2 gives a motivating example. Section 3 presents our approach. Section **??** describes the implementation details underlying our approach.

2. EXAMPLE

We next illustrates how our techniques helps performance analyst to navigate and do drill down into performance problems. As an example, we use a trace file[1] collected via TPTP. Figure 1 shows the relevant parts of the trace file.

The trace file consists of several elements. `TRACE` is the root element for a valid document in XML. The `node` element contains the

[1]http://www.eclipse.org/tptp/platform/documents/resources/profilingspec/trace.dtd.html

```
<?xml version="1.0"?>
<TRACE>
<node nodeId="..." hostname=""
 ipaddress="" timezone="" time="..."/>
<processCreate processId="..." pid="..."
 nodeIdRef="" time="..."/>
<agentCreate agentId="..." version="..."
 processIdRef="..." agentName="Java Profiling Agent"
 agentType="Profiler" agentParameters="server-enabled"
 time=""/>
<traceStart traceId="..." agentIdRef="..."
 time="..."/>
<filter pattern="..." mode="..." .../>
<option key="..." value=""/>
<threadStart threadId="..." time="..."
 threadName="Thread-38" groupName="main"
 parentName="system" objIdRef="0"/>
<threadEnd threadIdRef="..." time="..." />
<classDef threadIdRef="..." name="..." classId="26359"/>
...
<methodDef name="doGet" signature="" startLineNumber="..."
 endLineNumber="..." methodId="26342" classIdRef="26359"/>
<methodEntry thredIdRef="" time="" classIdRef="26359"
 methodIdRef="26342"/>
<classDef threadIdRef="..." name="..." classId="26361"/>
<methodDef name="create" signature="" startLineNumber="..."
 endLineNumber="..." methodId="26345" classIdRef="26361"/>
<methodEntry thredIdRef="" time="" classIdRef="26361"
 methodIdRef="26345"/>
<methodExit threadIdRef="" time=""...>
...
</TRACE>
```

Figure 1: Example Trace file

information that is associated with a specific node or host. The `processCreate` element... The `classDef` defines a class and `methodDef` defines a method, respectively. The `methodEntry` contains the information that is associated with a method entry. Especially, it has `methodIdRef` and `classIdRef` attributes. If the value of the attribute `methodIdRef` and `classIdRef` are the same as `classId` of the previous `classDef` and `methodId` of the previous `methodDef`, we identify the `methodEntry` elements as the child of the transaction.

For J2EE-based web applications, when a url request arrives at the server side, the web application invokes `doGet` or `doPost` method to process the url request. So, we treat the methods as the beginning of a new transaction. For example, there is a `classDef` that its `classId` is `26359`. The `classIdRef` of the `methodDef` named `doGet` is `26342` that is the same as the previous class definition. So, the method is the starting point of a new transaction. Because `classIdRef` and `methodIDRef` of `methodEntry` are `26359` and `26342` respectively, we identify the element as a child of the transaction.

3. APPROACH

The high-level overview of our approach is shown in Figure **FIX**prepare for the Figure**FIX**. Our approach consists of two major phases: Correlating Trace Data and Exploiting Statistically Sampled Trace Data. In the first phase, we focus on correlating (fully-collected) traces from the server side with the client-side performance test data without requiring the involved servers to be ARM-instrumented. In the second phase, we focus on using statistically sampled traces in the techniques from the first phase instead of full trace data.

3.1 Correlating Trace Data

In order for performance analyst to navigate from the client-side runtime information to the server-side runtime information in performance analysis, correlating client-side transactions or time-ranges with server-side extrapolated runtime information should be done. For an ARM-instrumented server, we explore ARM hooks

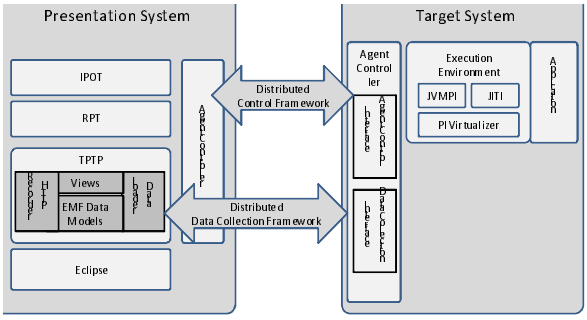

Figure 2: The high-level overview

for providing the information on when transactions begin and end. Each ARM instrumented server generates a correlator to map parent transactions to their respective child transactions across multiple processes or servers. Thus, the correlation mechanism can trace the path of a distributed transaction through multiple processes or servers. We further correlate with the runtime information collected via JVM profiling with the information collected via ARM hooks; therefore, we can establish the correlation between the JVM profiling information and the information from the load testing tool at the client side.

Next, we show how to correlate the traces **FIX**add more explanations**FIX** A model generated by RPT is in the format of Eclipse Modeling Framework (EMF[2]) model. As the traces generated by TPTP are stored as EMF model, the model generated by our approach is also an EMF model.

In case of absence of ARM, we develop learning techniques to discover with transactions begin and end at the server code. There will be three phases in the learning process: training, validation, and analysis.

3.2 Exploiting Statistically Sampled Trace Data

4. EVALUATION

Our evaluation is ...

5. RELATED WORK

Jinsight [3] is a tool for visualizing and analyzing the execution of Java programs. **FIX**Comparison will be here.**FIX** Liang and Viswanathan [5] presented a general-purpose, portable, and extensible approach for obtaining comprehensive profiling information from the JVM. **FIX**Comparison will be here**FIX**. Atom [8] is a programmable instrumentation toolkit. **FIX**Comparison will be here**FIX**.

6. CONCLUSTION

Our conclusion is ...

7. REFERENCES

[1] Application Response Measurement. `http://www.opengroup.org/management/arm/`.

[2] Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.

[3] Jinsight. `http://www.research.ibm.com/jinsight/`.

[4] Java Virtual Machine Tool Interface. http://www.j2ee.me/j2se/1.5.0/docs/guide/jvmti/.

[5] S. Liang and D. Viswanathan. Comprehensive profiling support in the javatm virtual machine. In *COOTS'99: Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems*, page 17. Berkeley, CA, USA, 1999.

[6] HP LoadRunner. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17^8_4000_100__.

[7] IBM Rational Performance Tester. http://www-01.ibm.com/software/awdtools/tester/performance.

[8] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4): 528–539, 2004. ISSN 0362-1340.

[9] Test and Performance Tools Platform. http://www.eclipse.org/tptp/.