University of Alberta

Detecting and Diagnosing Web Application Performance Degradation in Real-time at the Method Call Level

by

Mengliao Wang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2012
Edmonton, Alberta

Abstract

As e-commerce becomes more popular, the performance of enterprise web applications becomes an important and challenging issue. Unlike failures, performance degradation is more difficult for the administrator to observe. It also takes much time to locate the root cause because there are many possible performance bottlenecks including network I/O, resource starvation, source code mistakes and even high site requests.

In this thesis, we propose a system which is able to detect and diagnose web application performance degradations at the method call level in real time. We also set four sensors on the web server to monitor web application conditions, and use the measurements as input in detection and diagnostic algorithms. We implement our system in Java and build a test bed using a pet store application. Our system is evaluated on the test bed and results are encouraging, showing that our approach outperforms a traditional detection approach.

Acknowledgements

First I would like to express my thanks to my supervisor Dr. Kenny Wong for his guidance and patience during the last two years. It would not be possible to finish this thesis without his support and help. I would also like to thank my lab mates Haiming Wang and Liang Huang for their feedback and suggestions. Last but not least, I sincerely thank my parents who supported my study over the last years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, an ever-increasing number of businesses publish and update their products and services through web applications. These web applications are required to be robust and responsive to process the requests from clients. As a result most web applications are hosted on powerful server machines to keep them running normally. However, in commercial uses the performance of web applications could still be degraded due to various reasons. For example, a web application server may receive a huge number of requests in a short period of time, which can greatly increase the response time for each client of the server. Also, the consumption of operating system resources like CPU, memory, and disk space on the web application server can also lead to performance degradation.

It is important for the administrators of web applications to detect the performance degradations once they happen, and fix these problems as soon as possible. However, it is impossible to monitor every client's running state due to the huge number of clients and privacy issues. Thus, some performance monitoring tools have been developed to help the administrators to detect performance degradation, such like CA Wily [3], HP Performance Center [6] and Glassbox [5]. All of these tools aim at application performance monitoring, but are different in the details. Glassbox uses AOP (Aspect Oriented Programming) techniques to monitor an application's health in real time. It monitors the requests as Java code executes and provides details about response times. The Glassbox web client provides a nice dashboard view which contains various attributes like server-name, application name, operation/request-URL, number of executions, status (slow/OK), and so

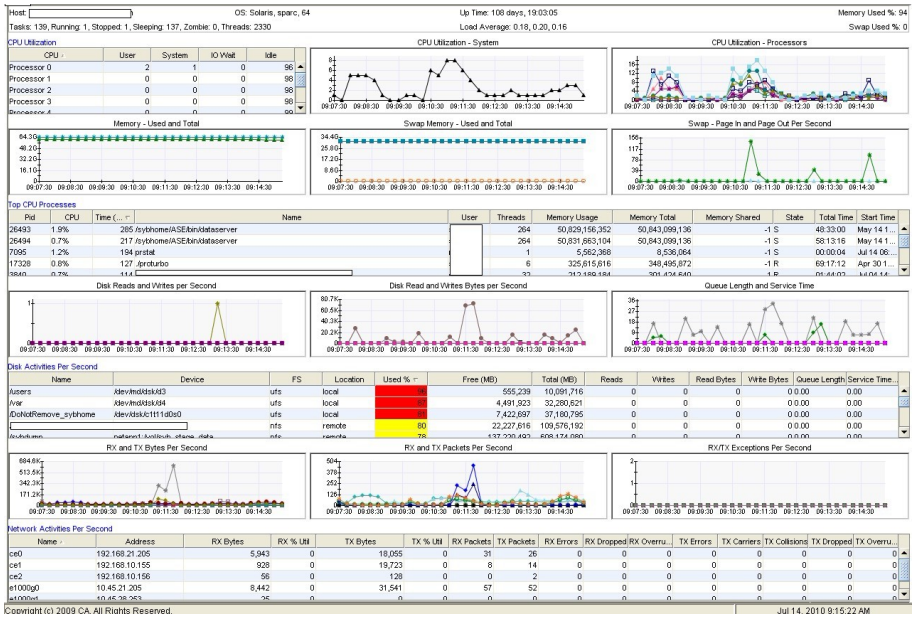

Figure 1.1: CA Wily Dashboard Example

on. But the Glassbox monitoring scheme is simple: by default, an operation that takes more than one second of execution time is marked with a "slow" status. CA Wily application performance management is another tool which provides performance monitoring functions. Similar to Glassbox, it provides a dashboard with details about an application's running status. Figure 1.1 shows an example of the dashboard provided by CA Wily. However, CA Wily does not support automated performance analysis, which means it can not detect when the application performance has degraded. HP Performance Center is similar to CA Wily, which provides a dashboard to present details on web application performance. But this tool still needs the experience and expert knowledge of the administrator to make the decision about whether there is a performance degradation and what the root cause of the degradation might be.

It is hence necessary to develop a more powerful tool to detect performance degradations of web applications and diagnose the causes, which is also the focus of this thesis. In this work, we propose a framework to detect and diagnose the performance degradations at the method call level with assistance from some sensors which monitor certain server attributes like CPU usage, memory usage, I/O rate, and request rate. Unlike the degradation detection scheme used by Glassbox,

we first collect enough baseline web application performance data, from which the distribution model of the execution time of each method under normal situations is calculated. Also, we mine the most frequent method call patterns from collected method call traces and save them with their execution time in a pattern repository for future analysis. If a method call trace is given, we use the pattern repository to divide it into segments, with each segment as the basic unit for analysis rather than the method. If one segment's execution time deviates too much from the normal execution time stored in the pattern repository, an alert will be given to the administrator.

Our approach not only detects the performance degradation, but also determines the potential cause of it. This is based on some server environmental status such as CPU usage, memory usage, I/O operation number, and the request number. The problem determination is based on some pre-defined status checks and eventually the problem will be categorized to be low CPU resource, low memory resource, high I/O rate, or high request rate. Moreover, if the problem is inside the web application, our approach can locate the method in the source code that reveals this performance degradation, which could greatly reduce the time for programmers to remove the fault. Most of the current performance monitoring tools only provide the administrator a dashboard with system performance information, but the performance problem detection and determination is still up to the administrators. In our research, the web application monitor could present an alert by itself of performance problems without human intervention. Also this framework can provide helpful suggestions during the problem fixing stage, and make performance tuning easier for the administrator.

The contributions of this thesis are:

1. Proposed an AOP based web application instrumentation system with low overhead and no need to recompile programs.

2. Proposed a method call trace based web application performance degradation detection framework which runs in real time. This approach is shown to outperform the detection approach used by Glassbox.

3. Proposed a performance degradation diagnostic system to determine the category of performance problem. Also this system is able to precisely locate the method in the source code that reveals the performance degradation.

4. Implemented our approach in Java, and built a test bed using the Java Pet Store application to evaluate our system, and the results validate the efficiency and effectiveness of our approach.

The remainder of this thesis is organized as follows. In Chapter 2, we describe the background of web application performance monitoring. In Chapter 3, the framework of our system is illustrated, and we detail each analysis phase of our system separately. In Chapter 4, we describe the evaluation details and present our experimental results. The analysis of the results demonstrates the efficiency and accuracy of our approach, and the overhead of our approach is shown to be low. In Chapter 5, we introduce some work by other researchers that overlap with our work. Finally, in Chapter 6, we summarize our contribution in this research and discuss potential future work of this topic.

Chapter 2

Background

Web applications have been an ever-increasing topic in e-commerce business during the last decade. A web application is hosted on a powerful server to provide services such as online shopping, web mailing, and online chatting for clients. On the client side, users always wish responsiveness from the server. For example, in an online chatting application, if the message received from the server is delayed, the conversation could be misunderstood or difficult to continue. However, in commercial applications, there are often a large number of users connected to the server simultaneously, which means the processing by the server might intensify at certain moments. Under this situation, the performance of application will be greatly slowed, which means each user will need to wait for a longer time than typical to get a response from the server after an operation.

In order to avoid such incidents, the administrator needs to monitor the running status of the web application, and detect the performance degradation as soon as possible. To assist in performance monitoring, some tools have been developed to provide administrators a dashboard with detailed server running status. CA Wily, Glassbox, HP Performance Center are such tools. With all the graphs and data from these tools, administrators can determine whether there is a performance problem and the cause of the problem. However, these tools still require human attention to make a decision about whether the server performance is good or degraded. Thus, in this thesis, we propose an approach for automatic performance degradation detection. This chapter overviews the web application tiers, and then describes the definition of performance degradation together with some potential causes. After

that, we introduce several existing state-of-art web application performance monitoring tools.

2.1 Definition of Performance Degradation

A typical web application has three tiers: web tier, application tier, and integration tier [13]. The web tier can be understood as the user interface in a web application, which supports the interaction with a human. In a J2EE (Java 2 Enterprise Edition) web application, JSP (Java Servlet Pages) is often used to implement this tier. The application tier, or business tier, is where the source code and pagers reside. The main task of this tier is to deal with the business logic of the application, which determines the functions this application supports. In J2EE, the application tier is often implemented by EJB (Enterprise Java Beans) and JavaScript. The integration tier, also called the data access tier, connects the stored dataset with the system. For example, in an online Java shopping application, the information on all the products may be stored using Derby [10], a database implemented in Java. When a user sends a request for searching a certain product from the web tier to the application tier, a connector such as JDBC (Java Database Connectivity) will connect the database to the application at a certain port. After that, the product could be searched by running the business logic upon the database. In a three-tier application, the transmission between different tiers takes time because these tiers are often not executed within the same process, or even the same machine. Also some components inside the tier need to take more time to execute. If these time-consuming components are repeated too often or their execution time greatly extended, a slower response results for the user, which we deem a performance degradation.

In the white paper by Tealeaf Technology [28], they propose that application health is not simply evaluated in terms of uptime and page load speed, but by incorporating other important metrics and the perspective of the user. This idea motivates us to monitor performance based on different metrics and sensors. One potential metric is method execution time. As we introduced before, if a performance degradation in the web application happens, there may be some abnormally

time-consuming methods. Using a method call trace with time recorded for each call is one way to detect performance degradations.

2.2 Causes of Performance Degradation

Performance degradation might happen in all the three tiers of a web application. We classify causes of performance degradations on the web server based on these three tiers with an additional category which is environmental causes. The details of potential causes we focus on are listed as follows.

2.2.1 Web Tier Causes

1. High request rate: There are too many users sending requests to the server in a short period of time, which causes the server response time to slow greatly for each client.

2.2.2 Application Tier Causes

1. Deadlock: In the business logic, a deadlock appears which will lead the application to be stuck for a long time until the deadlock is fixed.

2. Thread consumption: Most operating systems have a maximum thread number, while some web applications need to create different threads for different users. Scheduling many threads may cause some users to wait for a long time.

2.2.3 Integration Tier

1. Database queries: Database searching is also a time-consuming process in web applications. If the database is referred to too frequently, the response time will increase.

2.2.4 Environmental Causes

1. Low CPU resource: On the web server, if the CPU is too busy to process the web application request, the response time of each request will increase.

2. Low memory resource: Similar to low CPU resource problems, if there is limited available memory usage, it will take more time for the web application to process the request from the client.

3. Low disk resource: If the disk resource of the web server is consumed without leaving enough disk space, some operations like database exchanges will be slowed.

4. High I/O rate: I/O consumption in the web application includes various types such as network I/O, database I/O, file I/O and so on. Massive I/O operations of any type in a short period of time can lead to a performance degradation in the web server.

5. Monitoring overhead: Processing time needed by the application monitoring component could increase the application response time.

2.3 Performance Instrumentation Approaches

2.3.1 Aspect Oriented Programming

For any application monitoring tool, the first step is to instrument the web application. One widely used instrumentation technique is called AOP (Aspect Oriented Programming) [22]. AOP is designed to separate concerns, making each component more clear. Each system can be decomposed into core concerns and cross-cutting concerns. The core concern usually captures the central functions of the system. The cross-cutting concerns may be authentication, logging, storage management, multithread safety, and so on. These concerns often affect other concerns and can not be cleanly decomposed from the rest of the system. Thus, by OOP (Object Oriented Programming) techniques, it is difficult to add, remove, or update the components which implement cross-cutting concerns in a system. However, in a typical AOP project, each cross-cutting concern is encapsulated in a separately implemented "aspect". An aspect is connected by a "pointcut", which specifies at which locations this aspect will be integrated into the system. This stage is called weaving. The process is shown in Figure 2.1. When a set of requirements is presented, the AOP technique classifies these requirements into core concerns and cross-cutting concerns, and implements each concern individually as an aspect. Note that all the cross-cutting concerns should be independently implemented. For example, updating the logging aspect should not affect the security aspect. Finally all the aspects are woven into the final system by the pointcuts.

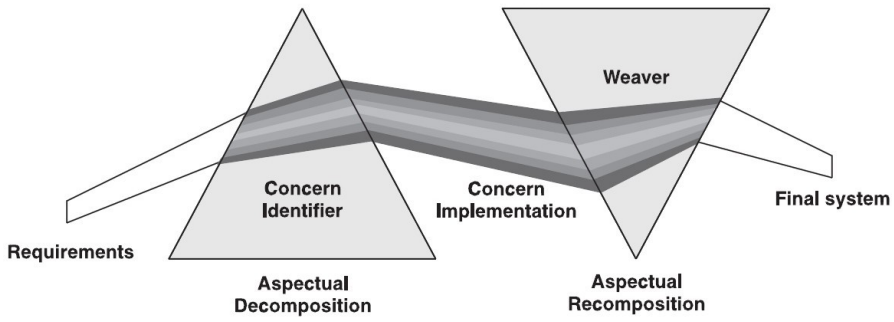

Figure 2.1: AOP Development Stages [22]

Because AOP makes it very easy to insert additional functions to the existing code at specific points, it is one of the most popular forms of instrumentation. Programmers need only to write a separate instrumentation aspect, and weave the instrumentation code into the original application source code. The original methods to be instrumented will be specified in the pointcuts. Thus, to instrument any application, programmers do not need to be familiar with the original application source code, and the instrumentation code can also be applied on different applications without too many modifications.

There are several tools that implement AOP, and the most popular one on Java is AspectJ. AspectJ is an extension of Java, which means every Java program is also a valid AspectJ program, and all the AspectJ programs are written in Java with some special AOP rules. AspectJ is a mature language used widely nowadays, but a major problem is it requires recompiling the whole software if we need to introduce some new aspects. In a big project, this process is very time-consuming and potentially dangerous. There is another AOP implementation, called Aspectwerkz [2], which does not require recompilation if new aspects need to be integrated. It uses a deployment description file to specify the weaving details and it is very simple for the programmers to use.

2.3.2 Application Response Measurement

ARM (Application Response Measurement) is an API (Application Programming Interface) to capture vital information about transactions inside applications or middleware at runtime. A transaction is a set of method calls which completes a certain function or action in the application. ARM is called before a transaction begins and stopped when a transaction ends. This technique is repeated for every transaction until the web application itself stops running. This API was implemented in C language only, without support for Java before 2002. In 2003, the Open Group allowed both C and Java bindings in ARM 4.0, which were compatible with each other.

When the application is running, each time a transaction is finished, ARM reports the information collected to a software management system. However the measurements about a transaction it supports are limited, including only response

time, running total time, and bytes transferred. Although this API provides many built-in transaction analysis functions, the lack of other measurements limit further analysis on application performance bottlenecks [20].

2.3.3 JavaSysMon

JavaSysMon is a simple API developed by J. Humble [9]. This tool is written in Java, with a dynamic linked library written in C. It is a cross-platform tool and is able to monitor different attributes of the operating system. This tool was developed because a Java application running on a virtual machine is unable to obtain certain operating system measurements directly and quickly. Thus this API assists Java programmers to monitor operating system conditions efficiently without high overhead.

This API is still under development, and currently it captures values such as uptime, CPU usage and information, total and free memory, as well as process table information with ID, name, size, resident size, user time, and kernel time for each process. The main advantage of this API is the low overhead compared to other instrumentation tools, which is very important in long-running commercial web applications.

Chapter 3

Methodology

When performance degradation happens to a web application, our tool is able to detect the occurrence and analyze the cause of this performance problem using information at the method call level. If the cause lies in the web application itself, our approach can locate the method of the web application that reveals this problem. In this section we describe the details of our web application performance monitoring and diagnosis system. Our approach includes both offline and online parts as shown in Figure 3.1 and Figure 3.2 respectively. In offline part we instrument the execution time of every method when the web application is running. We collect a number of normally running traces to mine patterns for trace abstraction and build models for each pattern, which assist our performance degradation detection algorithm. The online part involves three phases: application instrumentation, performance degradation detection, and cause diagnosis.

1. Application instrumentation: Each web application consists of a number of method calls, and we abstract the method call trace into a set of patterns. When a method call begins, we first detect whether it is the end of a pattern we mined in offline stage. It yes, our instrumentation system starts to collect information about the application running status and server environment parameters from four sensors: CPU usage, memory use, I/O rate, and request rate. This information is passed to the next phase.

2. Performance degradation detection: When a pattern is found, the information during the execution of this pattern is collected. The second phase makes a

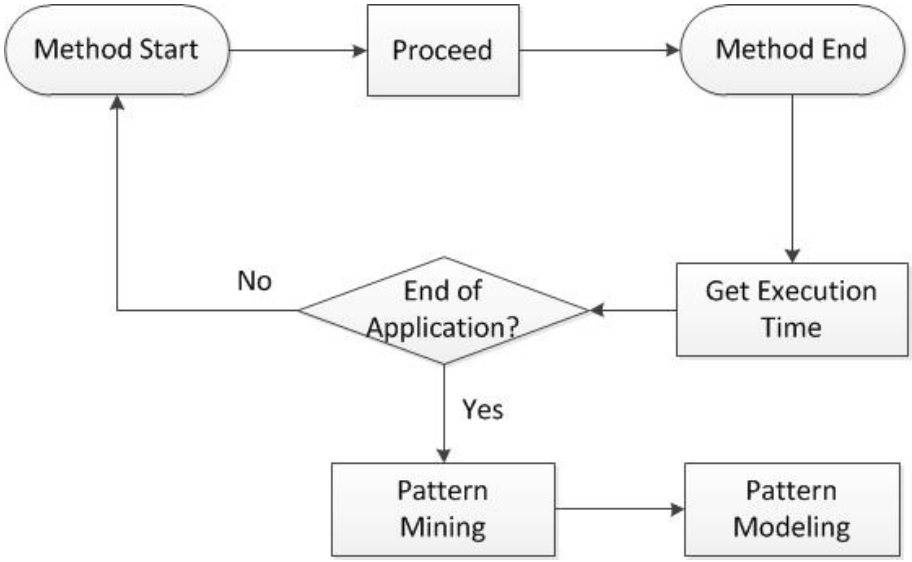

Figure 3.1: Architecture of the Offline Stage

decision about whether performance degraded at the end of this pattern based on the information collected. If the answer is yes, we proceed to the third phase to categorize the cause of this performance problem. Otherwise we continue monitoring the next method call.

3. Cause diagnosis: If a degradation is detected, the third phase determines the category of this cause. This phase is also able to locate the method in the source code that reveals the performance degradation. Administrators can then use the suggested method as a starting point when debugging.

3.1 Offline Stage

Given that our instrumentation system is at the method-call level and the data collected are affiliated with method calls, we need to organize and abstract the method call sequence for further analysis. Otherwise the information will be too huge to process in real-time. A very simple operation by the user, like adding a product to cart, may lead to hundreds of method calls with each method call accompanied by a piece of system information. Here, we segment the method call trace and find some common patterns for abstraction. Next we select patterns, which are healthy since no performance degradation is happening, and use these healthy patterns to

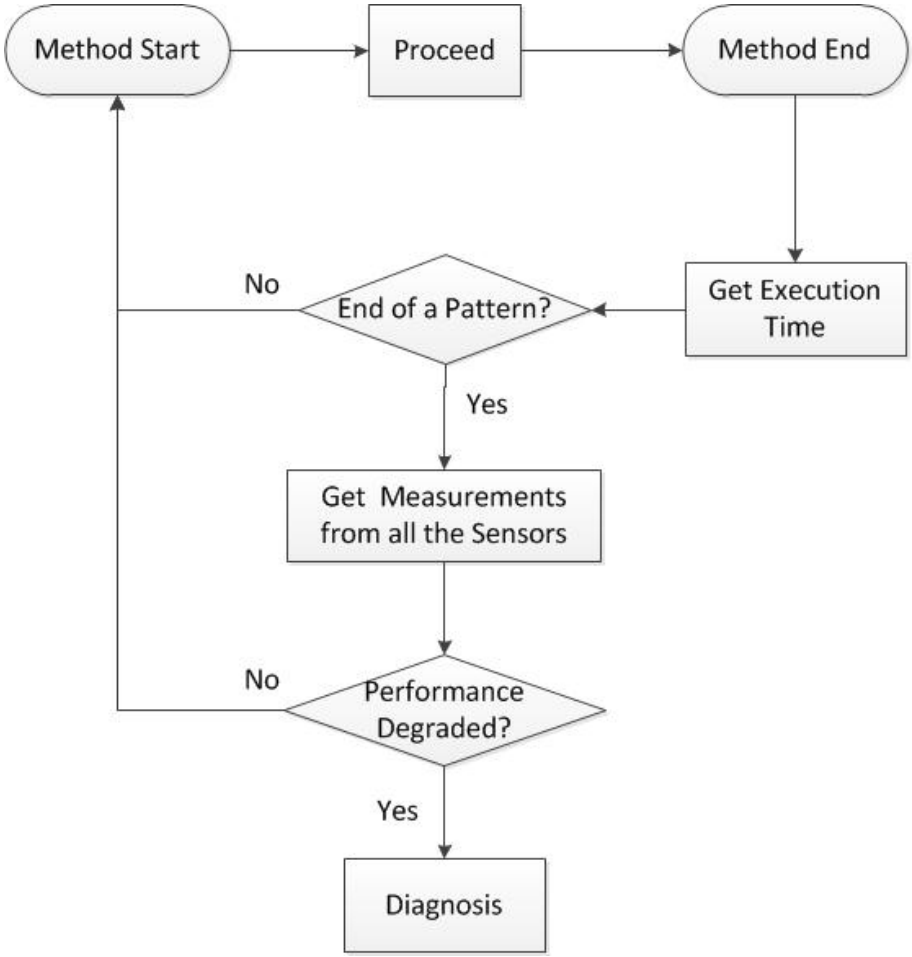

Figure 3.2: Architecture of the Online Stage

build a model of pattern execution status. When our system is running and detecting performance problems in real-time, our system can determine the probability of whether there is a performance degradation happening inside the current method call trace pattern based on the built model.

3.1.1 Pattern Mining

In a complex commercial web application, the size of a method call trace is often huge. In our test bed, which is built based on a demo pet store application, about 5,000 method calls are involved in a typical shopping process. As a result trace abstraction is needed if we want to analyze the web application situation at the method call level. During the last decade, a lot of research has been done on method call trace segmentation [15, 27, 25, 30, 17]. The most commonly used approaches

are based on pattern selection. K. Sartipi et al. [25] used some machine learning algorithms to select a set of most representative patterns, and used these patterns as the basic unit to process in later analysis. In our tool, due to the speed and space limitations, our algorithm must not be too complicated and be able to finish quickly.

The method call trace is a stack structure, which means if method A is called before method B, method B must end before method A ends. There is no exception on this calling sequence. Also, certain methods may be called repeatedly which is usually caused by logical loops in the application source code. These loops cause the trace file and the system information recorded at each method call to be large and repeating. Thus there is a pre-processing step in our detection system to remove redundant method calls caused by loops in the web application.

Each time a method is finished and recorded by the instrumentation system, in order to check whether this is a repetitive method call we consider the k previous method calls. As an example, suppose sequence "$AA'BB'CC'BB'$" is the method call trace so far, which we assume to contain no continuously repetitive method calls. Here symbol "A" refers to the method entry, while symbol "A'" refers to the method exit. If method A throws an exception, it is also labeled as "A'". When next method call occurs, we apply a repetitive string finder algorithm (Crochemore [16]) on the recent calls trace to see if the incoming method call leads to a repetition. If yes, e.g. the next method is "CC'" or "BB'", the repeated method calls are composed as one instance with the number of repetitions labeled, such as "$AA'(BB'CC')_2$" or "$AA'BB'CC'(BB')_2$". Otherwise, e.g. the next method is "A", we do not have a repetition.

However, even if we group the repeated method calls as one instance, it is still slow to collect performance information at every single method call. In our experiment we find that some sensors like memory usage and free disk space are very time consuming and even take longer time than some web application methods. This can increase the overhead on each instrumented method and slow down the web application greatly. This is also a reason why approaches like Glassbox use only a simple sensor like method execution time on each method call to detect performance degradation. To reduce the overhead generated by multiple sensors,

we apply trace abstraction techniques on the method call trace, and detect web application performance degradation based on each segment rather than each single method call.

We show the pseudocode of our abstraction algorithm in Figure 3.3. At first, to do trace pattern mining, we need a set of normal method call traces without performance problems for training. Then we use a sliding window with different sizes to scan the training traces. The maximum window size s is set to be 50 in our experiment. The window size can differ but must be a even number because each method consists of an entry/exit pair. For each fixed size window, we scan through the whole method call trace. Every time we move the window one step forward we will get a new subsequence, and this subsequence needs to pass a validation test to be selected as a pattern. The validation rule is that the entry/exit pair for the same method call must be found in the subsequence. That means, if method A's entry is in the subsequence, then method A's exit must be in the subsequence too, and vice versa. For example, segment "$AA'BCC'B'$" is a valid segment, but segments "$ABB'CC'D$" and "$AA'BCDD'$" are not valid. This rule guarantees that each subsequence is a separate unit and complete several functions. The whole method call trace could be decomposed into these subsequences.

PatternMining(T, LENGTH, MAX_SIZE) : PR
Input: A method call traces T, the length of trace LENGTH, a maximum sliding window size MAX_SIZE
Output: a pattern repository PR

```
size = 2
pos = 0
while size <MAX_SIZE do
    while pos <LENGTH do
        Run pattern validating algorithm
        if T(pos : pos + size) is a valid pattern then
            Add T(pos : pos + size) to RP
        end ifpos = pos + 1
    end while
    size = size + 2
end while
return RP
```

Figure 3.3: Pattern Mining Pseudo Code

All the subsequences that follow these rules are picked offline and denoted as patterns, and the patterns are stored in a pattern repository together with their normal execution time. We use these patterns as the basic units to handle rather than the single method calls. When our application is running, at each method call we

first scan the pattern repository to see if this is the end of a pattern. If yes, we collect the system information from all the four sensors, and then start the performance degradation detection. Otherwise we skip this method call and continue. However, the size of the pattern repository is usually too huge to handle if all the patterns are saved, so we need to select the most common ones and remove the patterns that appear less than n times. In our experiment we set n to be 50. Here if two patterns are almost the same except for some method repetitions, they will be recorded as the same pattern in the pattern repository. For example, Figure 3.4 shows two patterns with repetitions of "$DEE'D'$". These two patterns are defined as one pattern stored in the pattern repository.

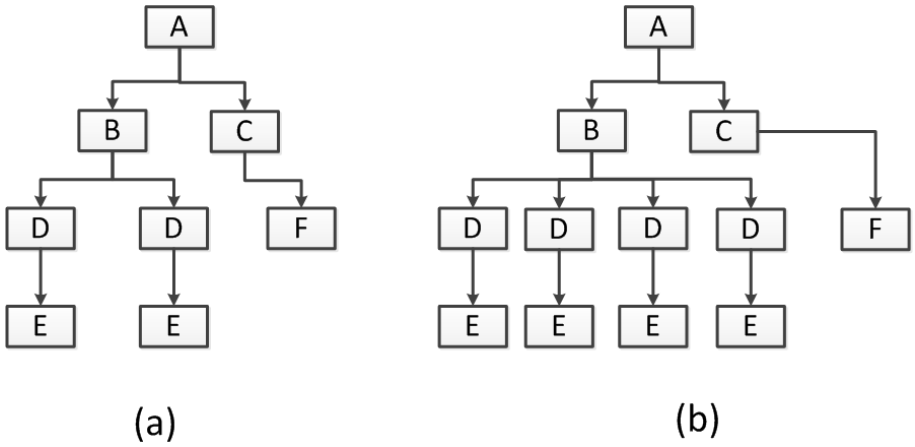

Figure 3.4: Example of two patterns with different numbers of repetitions

3.1.2 Pattern Modeling

As we discussed before, it is not a good idea to detect performance degradation by only setting a single threshold on execution time. Because the execution time distribution of each method varies greatly, the mean execution time per method varies too. In Figure 3.5, we give an example of the execution time of method calls under an injected performance degradation. At the beginning of this trace, some methods take a very long time to finish, even when there is no performance degradation there. These methods are time-consuming simply because of their task. Thus in this example, a single threshold approach cannot detect the performance degradation accurately.

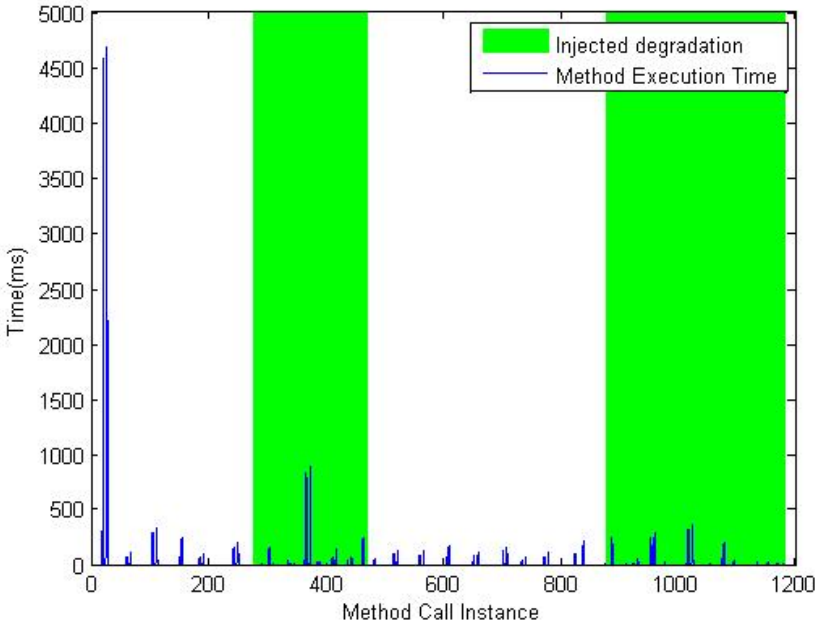

Figure 3.5: Example Method Call Trace with Performance Degradation

Here we propose an idea of treating each method differently. To be more specific, we collect enough execution time data for each method separately, and use this data to calculate the distribution parameters for each method, including mean value, standard deviation, and frequency. To make the modeling step easier, we assume that the method execution time distribution fits a Gaussian model. This assumption is validated in an experiment in Chapter 4. The mean execution time and standard deviation of method i are recorded as the Gaussian model parameters μ_i and σ_i. The probability that method i take x seconds to finish, based on data collected from a normally running system, is calculated by Formula 3.1. If we detect performance degradation by setting a threshold on the probability, the problem of using a threshold on absolute execution time is solved.

$$P_i(x) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{\frac{-(x-\mu_i)^2}{2\sigma_i^2}} \tag{3.1}$$

Because the method call trace is abstracted into a set of patterns, we model the patterns we mined rather than a single method. This is similar to method call modeling, but the execution time is over a number of method calls. To be more

specific, in offline training if a pattern is detected in the method call trace, we record the total time it takes to finish. If a pattern has repetitive loops, we use the average execution time of loops instead of the total execution time of all the loops. When all the patterns are saved in the pattern repository, the mean time and variance of each pattern are calculated and saved. There are two major benefits of detecting degradation based on the execution time of a pattern rather than a single method call. First, the execution time of the patterns varies less, i.e., lower variance in the Gaussian model than with method calls, which makes our approach more accurate. Second, in this way we can reduce the time spent on detection significantly. For example, if a pattern contains 20 method calls, we need to check for a performance degradation only when the 20th method call ends. In this way, we save a lot of time compared to running our detection algorithm at the end of every method call.

3.2 Online Stage

3.2.1 Application Instrumentation

In order to monitor the application performance at the method call level, the first step is to instrument each method separately. Methods in the three web application tiers usually function in different ways, which requires different instrumentation techniques to collect the parameters and health measurements of different method calls.

In this project we designed an instrumentation system for a typical J2EE web application. Figure 3.6 shows the architecture of this system. At first a user request initiated from the client side is sent to a web filter. We use a class RequestInstrumentation to collect the status of client requests. After that, the request is then processed by Java servlets. In most recent Java web applications, servlets are usually used in conjunction with JSP (Java Server Pages). Then the requests are forwarded to application functions which are commonly implemented by EJB (Enterprise JavaBean) components. Here, some methods may need to query data stored in the application database, so a DAO (Data Access Object) method is involved. These steps use similar Java standards and can be instrumented by one class, which we

name JavaInstrumentation. If a database is involved, we need another class DatabaseInstrumentation to record the database queries made. Because most databases are stored in certain files on the server and require I/O operations to read, this class will also monitor the related files. All these instrumentation classes also record the sensor measurements during the method execution phase.

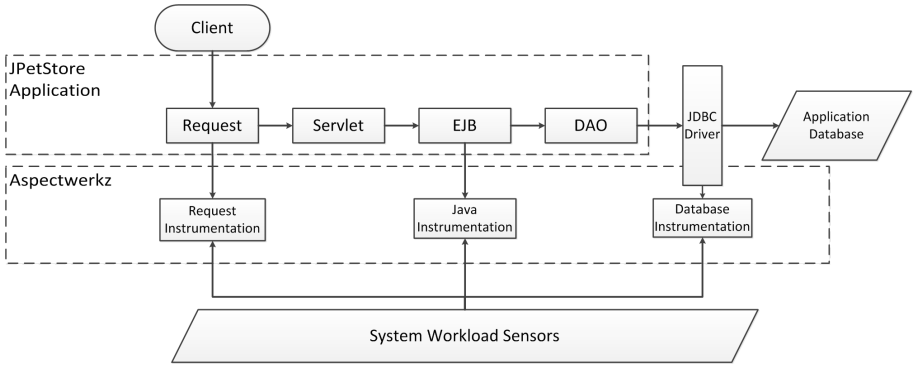

Figure 3.6: Instrumentation System Architecture

When the web application is running, at each method call, the server workload and method execution status are recorded by our instrumentation system with the details transferred to the detection phase for further analysis. Each method call is actually recorded as an entry/exit pair, and measurements are recorded separately for CPU usage, memory usage, total I/O operation number, and received request number. Note that CPU usage is actually a time-based statistic, which means that only the average CPU usage over a very short time can be estimated rather than the CPU usage at a certain time point. For example, the CPU usage from time A to time B is calculated using Formula 3.2. In this formula, "WorkingTime" refers to the time that the CPU spent in the busy state since the server was booted, and "TotalTime" refers to the time that the CPU spent in both busy and idle states. However, it is impossible to acquire the exact CPU usage at time A or B.

$$CPU\_usage = \frac{WorkingTime(B) - WorkingTime(A)}{TotalTime(B) - TotalTime(A)} \tag{3.2}$$

Also, the execution time of each method is defined to be only the time spent in the method excluding the time for all the sub-methods called by this method. For example, suppose method B and method C are called by method A, which results

in a sequence as shown in Figure 3.7. When we need to calculate the execution time t_A of method A, we will have equation $t_A = (Ex(A) - En(A)) - (Ex(B) - En(B)) - (Ex(C) - Ex(C))$. Here En refers to the time when a method starts and Ex refers to the time when a method ends. We exclude the running time of the called methods to pinpoint performance problems for each method more precisely.

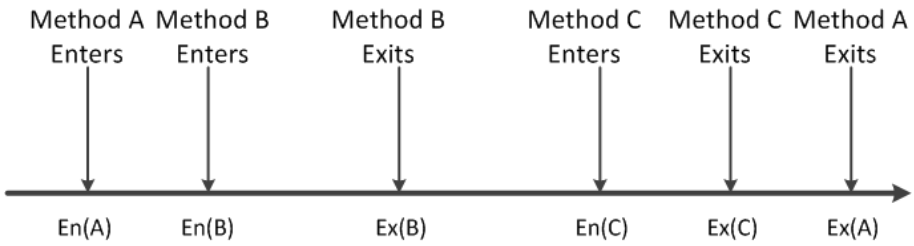

Figure 3.7: Method Call Sequence Example

As an example, Figure 3.8 is an excerpt of our recorded method call trace with sensor information. The trace is logged based on time series, and system information is captured at the exact point when a method entry or exit happens. Contents separated by colons from left to right at each line are: method call number, programming language, method name, start (for method entry) or stop (for method exit), system time (millisecond), time used to finish a method (millisecond), free memory, CPU usage, received request number, and total I/O operation number. Here the sample trace is used to represent what our instrumentation system logs. Given that our performance monitoring tool is running in real time, to speed up processing and save space, this information is transmitted directly to the diagnosis phase without logging.

3.2.2 Performance Degradation Detection

In order to decide whether performance degradation is happening, a system is needed to detect with high precision and recall based on the observation of server measurements captured by the instrumentation. This is also the prerequisite for diagnosing the application performance problem in the third phase. However, this is a difficult task given that application performance is affected by numerous factors which cannot be monitored completely. We use different metrics to evaluate the

```
1  28:JAVA:com.ibatis.jpetstore.service.CatalogService.getCategory
       :START:1323905835835:0:646520832:0.48387098:12:31
2  29:JAVA:com.ibatis.jpetstore.persistence.sqlmapdao.
       CategorySqlMapDao.getCategory:START
       :1323905835839:0:646520832:1.0:12:31
3  30:JAVA:com.ibatis.jpetstore.domain.Category.setCategoryId:
       START:1323905835889:0:646447104:0.8387097:12:37
4  30:JAVA:com.ibatis.jpetstore.domain.Category.setCategoryId:STOP
       :1323905835892:3:646447104:0.8387097:12:37
5  31:JAVA:com.ibatis.jpetstore.domain.Category.setName:START
       :1323905835895:0:646447104:0.8387097:12:37
6  31:JAVA:com.ibatis.jpetstore.domain.Category.setName:STOP
       :1323905835896:1:646447104:0.8387097:12:37
7  32:JAVA:com.ibatis.jpetstore.domain.Category.setDescription:
       START:1323905835899:0:646447104:0.5:12:37
8  32:JAVA:com.ibatis.jpetstore.domain.Category.setDescription:
       STOP:1323905835901:2:646447104:0.5:12:37
9  29:JAVA:com.ibatis.jpetstore.persistence.sqlmapdao.
       CategorySqlMapDao.getCategory:STOP
       :1323905835903:64:646447104:0.5:12:37
10 28:JAVA:com.ibatis.jpetstore.service.CatalogService.getCategory
       :STOP:1323905835906:71:646406144:0.5:12:37
```

Figure 3.8: Excerpt of Our Recorded Method Call Trace

running status of an application, but according to Tealeaf Technology's white paper [28], method execution time is the most direct and reliable one to detect performance degradation due to the causes we list in Section 2.2. In this thesis, the other metrics are only used for the diagnosis phase.

As we described in Section 2, the approach used by Glassbox simply set a single threshold on the method execution time to decide whether there is a problem within the application. However, due to the variety of the causes on the client side that might trigger a slow response, it is extremely difficult to decide if it is a performance degradation on the server that leads to this slow response or some other cause like internet connection failure. Also, based on our experiments we find that execution times of different methods vary greatly. For example, the average execution time of method getProductListByCategory in class ProductSqlMapDao is 227.8ms, while the average execution time of method getItemList in class CatalogBean is only 2.67ms. So obviously one cannot set a single threshold on method execution time to detect whether there is a performance degradation. Here we designed an improved approach for performance degradation detection. The data flow chart of

this phase is shown in Figure 3.9.

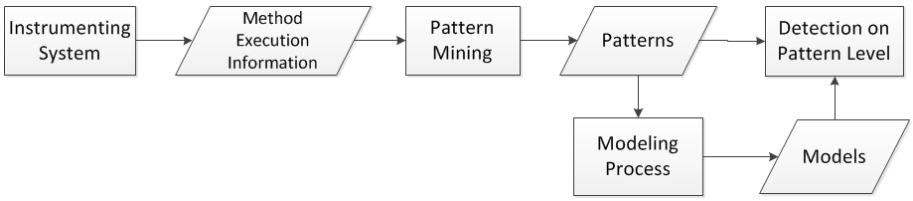

Figure 3.9: Data Flow of Performance Degradation Detection

Our instrumentation system records information about server status when each pattern ends, and then passes the data collected into the detection phase. We use the pattern repository and the Gaussian models created in the offline stage to detect unknown performance degradation in the online stage. When the application is running, our instrumentation tool will capture each method entry and exit in sequence. A pattern must be end with a method exit. When a method exit is captured by the instrumentation system, we search for the longest length match to the patterns in the repository. The longest length match searching process continues until a matched pattern is found, or there is no matches in the repository. If we cannot find a matched pattern, the method call is treated as a one-method pattern. Next we use the Gaussian model of the recognized pattern to calculate the probability that the execution time is normal. If the probability is greater than the threshold we set, this pattern is considered to be running normally. Otherwise, an alert will be presented which indicates there is a performance degradation suspected.

However, there is a problem about outliers of performance which is unpredictable in most cases. Even when the server is running in the best health condition, some methods might suddenly take much longer time than average to finish. The potential reasons vary and are difficult to predict, such as a transient network delay or a bad block on disk. The duration of this type of problem is usually only over 1 or 2 method calls. Compared to a performance degradation, which usually lasts for a longer period, we call transient cases as outliers. To solve this problem and improve the result, we use a filter to select the real performance degradation only. This filter is similar to the idea of median filtering in the signal processing area. In this thesis, the signal is a method call trace, while the signal value is the probability

of whether the current pattern execution time is normal. Each probability of the method call trace is recalculated as shown in Formula 3.3. Before we report the final probability of method i, we check if the probability is obviously deviated from the mean probability from method $i-k$ to method $i+k$. We set k to be 5 in our experiments, and the deviation threshold is decided by α which we set to be 0.4. If $P(i)$ is obviously deviated, we claim this is a outlier and replace the probability by the mean probabilities of $2k+1$ methods. Otherwise we do no adjustment. This filter is capable of specifically removing the outliers.

$$P(i) = \begin{cases} \frac{1}{2k+1} \sum_{j=i-k}^{i+k} P(j) & , |P(i) - \frac{1}{2k+1} \sum_{j=i-k}^{i+k} P(j)| > \alpha \\ P(i) & , |P(i) - \frac{1}{2k+1} \sum_{j=i-k}^{i+k} P(j)| < \alpha \end{cases} \tag{3.3}$$

3.2.3 Performance Degradation Diagnosis

If a performance degradation is detected and confirmed in the detection phase, the next step is to diagnose and identify the potential cause. However, this task involves system administrators in most web application monitoring tools. These tools only display the collected system data on a dashboard, but it is up to human operators to analyze and locate the cause. This is understandable because a lot of reasons can lead to performance degradation. It is extremely difficult to locate all the potential causes we listed in Section 2.2 and other uncertain causes with one general diagnosis system. In this case most researchers focus on certain types of performance problems, or use expert knowledge to diagnose these problems. In our research we set four different sensors on CPU usage, memory usage, I/O rate, and request rate to diagnose the following four causes: low CPU resource, low memory resource, high I/O rate, and high request rate. Our approach can locate the type of cause using information at the method call level, but only for the performance problems caused by the web application itself. Figure 3.10 illustrates the detailed steps to diagnose a confirmed performance degradation.

If performance degradation is detected in detection phase, we use the four sen-

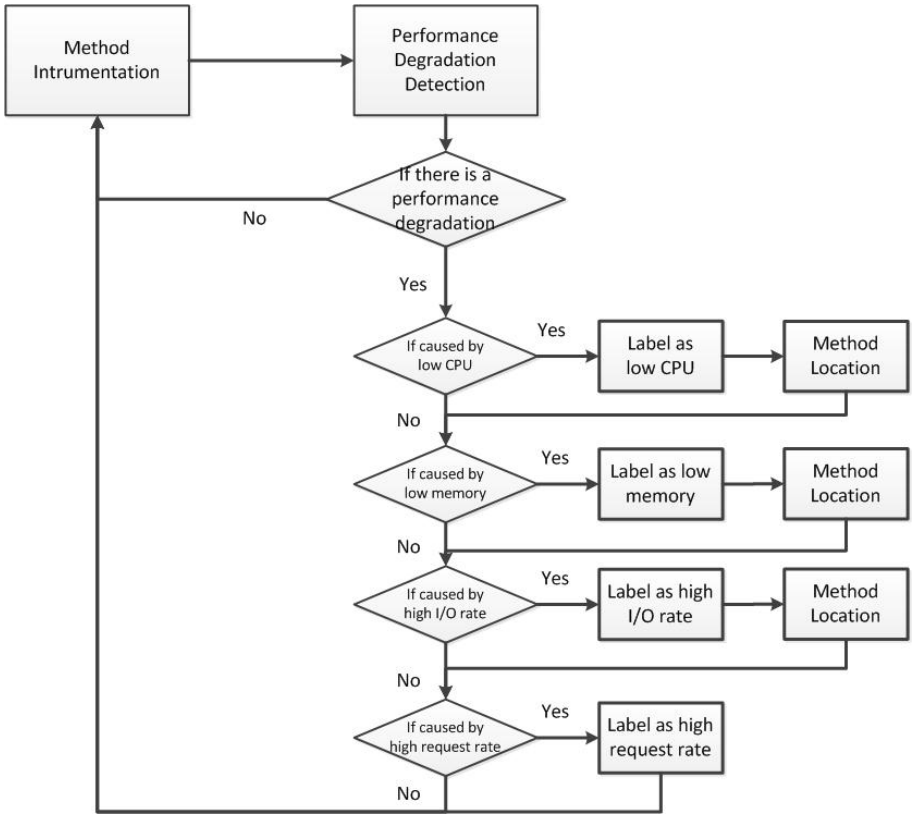

Figure 3.10: Degradation Diagnosis Framework

sors to classify the cause and the method where the degradation occurred. This method might suggest the real cause. If all the potential causes can not be confirmed based on the sensors we use, this problem is categorized as a undefined performance degradation and is left to the administrator. The details of each step will be described in the following sections.

Cause Classification

It is stated in Section 2.2 that a performance degradation can be caused by various reasons, such as problems in program logic, resources, hardware, or the network. All these causes have different performance effects on the server and require different approaches to determine, but on the client side they all appear to be the same symptom: it takes longer time to get a response from the server, even leading to a failure. We can determine whether there is some performance degradation based on method execution time. However, for performance degradation diagnosis, the ex-

ecution time of each method call is not enough and we need assistance from other sensors on crucial system attributes. The relationship between each system attribute is difficult to model. Thus we need to deal each sensor data separately. In our experiment we use four different sensors on CPU usage, memory usage, I/O rate, and request rate, aiming at the causes related to low CPU resource, low memory resource, high I/O rate and high request rate respectively.

Unlike method execution time, the state of resources such as CPU or memory may vary greatly depending on the time of day and the server condition. Thus we can not apply the same statistical strategy in the detection phase here. Instead of training models of system information offline, we use this information at runtime to determine the type of causes (see the pseudo code in Figure 3.11). First we consider the sensor data for the previous t pattern occurrences, and calculate the average sensor value over these t patterns. Next for each type of sensor, we compare the current value to the average value in this window. If the difference exceeds a threshold we set based on expert knowledge, we suggest a potential corresponding cause (e.g., low CPU resource for an exceeding CPU usage value).

We use this sliding window strategy to determine the type of cause because there is not an absolute expected value of each sensor. For example, in our experiment, we find that a web application might be working normally with 100 simultaneous requests, but can slow down when only 50 simultaneous requests are received. So we use the difference between the current value and recent values. Also, performance degradation might be caused by multiple reasons. For example, simultaneous request traffic may also lead to a low CPU resource symptom, so we check all the sensors.

Locating the Problem

Here, as a starting point to locate the root cause of the performance problem, we consider the method that revealed the performance degradation. The most direct way is to consider the method called at the time when performance degradation was detected. However, sometimes there is a delay between the time when the most suggestively problematic method is executed and the time when a performance

CauseClassification($cpu_{1,2,...,N}$, $mem_{1,2,...,N}$, $io_{1,2,...,N}$, $rr_{1,2,...,N}$,cpu_{N+1}, mem_{N+1}, io_{N+1}, rr_{N+1}):$V(P, C)$
Input: Saved system state including CPU usage cpu, memory usage mem, IO rate io, and request rate rr from pattern P_1 to P_N, system state of pattern P_{N+1} including cpu_{N+1}, mem_{N+1}, io_{N+1}, and rr_{N+1}
Output: array $V(P, C)$, P is the set of patterns and C is a 4-bit vector indicating the cause classification result

$mean\_cpu = \frac{1}{N}\sum_{j=1}^{N} cpu_j$
$mean\_memory = \frac{1}{N}\sum_{j=1}^{N} mem_j$
$mean\_io = \frac{1}{N}\sum_{j=1}^{N} io_j$
$mean\_rr = \frac{1}{N}\sum_{j=1}^{N} rr_j$
if $|cpu_{N+1} - mean\_cpu| > THRES\_CPU$ **then**
  $V(N+1, 1) = 1$
else
  $V(N+1, 1) = 0$
end if
if $|mem_{N+1} - mean\_memory| > THRES\_MEM$ **then**
  $V(N+1, 2) = 1$
else
  $V(N+1, 2) = 0$
end if
if $|io_{N+1} - mean\_io| > THRES\_IO$ **then**
  $V(N+1, 3) = 1$
else
  $V(N+1, 3) = 0$
end if
if $|rr_{N+1} - mean\_rr| > THRES\_RR$ **then**
  $V(N+1, 4) = 1$
else
  $V(N+1, 4) = 0$
end if
if application is running **then**
  CauseClassification($cpu_{2,3,...,N+1}, mem_{2,3,...,N+1}, cpu_{N+2}, mem_{N+2}, io_{N+2}, rr_{N+2}$)
else
  return V
end if

Figure 3.11: Cause Classification Pseudo Code

degradation is detected. For example, a performance degradation may be detected after method A, but could be due to numerous I/O operations in method B which executed 10ms before method A. One reason is the pattern-based approach: we typically check for a degradation after a number of method calls, not after each one. To address this issue, when a performance degradation is detected at method M, we list all the r methods happening before M together with M as suspicious methods. In our experiment we set $r = 20$. Next time if a similar degradation happens, i.e., with the same type of cause inside the web application, we list these $r + 1$ suspicious methods again for the suspicious methods list. For any further similar degradations we order the suspicious methods by frequency and suggest the

top ones as the method to start looking for the cause.

The parameters used in our approaches are listed in Table 3.1.

Table 3.1: Parameters Used in Our Approach

Parameter	Value used in the experiment
Maximum Pattern Size s	50
Pattern Occurrence Times Threshold n	50
Probability Threshold in Detection Phase p	0.1
Outlier Filter Deviation Threshold α	0.4
Outlier Filter Sliding Window Size k	5
Cause Classification Sliding Window Size t	10
CPU Usage Threshold $THRES\_CPU$	0.3
Memory Usage Threshold $THRES\_MEM$	0.2
I/O Rate Threshold $THRES\_IO$	5.0
Request Rate Threshold $THRES\_RR$	0.1
Method Location List Size r	20

Chapter 4

Evaluation

In the last few sections we described our framework on detecting and diagnosing web application performance degradation, and here we evaluate our approach against some approaches like Glassbox to see the effectiveness of our approach. In this chapter we first introduce the web application selected for testing and the implementation of the sensors. Then we illustrate how we inject performance problems into the web application for evaluation in our experiment. Next we describe the details of the evaluation design and conclude with results.

4.1 Test Bed

In our experiment, we choose a popular sample Java pet store application which was developed by the Java BluePrints program at Sun Microsystems. This application is designed to illustrate the functions Java Enterprise Edition 5 has, and also gives an example on how to design, compile, build, and run an e-commerce application. It implements web-based pet store shopping system, which contains functions including pet category browsing, user registration, logging in, shopping cart, specific pet search, check out, and retrieving shopping history. Here the version of the application we used is iBatis JPetStore 4 [7], which has 255 functional methods in the source code and uses a Derby database [10] to store product information. We choose this application based on the following reasons. First, this application is easy to compile and is built on Tomcat, a very stable web server [14]. It also means less conflicts and bugs when instrumenting this application using AOP techniques.

Secondly, the database of this application is open source, which means it is possible to inject problems into the full source code and recompile it. Finally, although this is a sample application, almost all the functions needed in real world e-commerce applications are well implemented in JPetStore and thus can mimic a real world scenario. This application is built and run on an Apache Tomcat Server (Pentium 4, 1 GB memory, 60 GB disk space). We can run this application repeatedly (reinitialization is also includeed) to set up different experiments.

4.2 Instrumentation

In this thesis we use Aspectwekrz [2] as our instrumentation tool. As we introduced in Section 2.3.1, Aspectwerkz is an implementation of AOP framework using Java. The most important feature of Aspectwerkz is that it does not need to re-compile the source code to add extra monitoring code. The monitoring code is embedded into an existing application through a deployment description file, so it is simple and clear for administrators to manage the monitored application. The deployment description file is based on XML schema and easy to understand. An example of our deployment file is given in Figure 4.1. The detailed information of each method execution is recorded in real-time, including system time spent, user time spent, total time spent, memory usage, CPU usage, I/O rate and request rate. Then this information is used for further analysis.

We use four sensors in this experiment, which are monitoring CPU usage, memory usage, I/O rate, and request rate. Because our instrumentation architecture is built on Aspectwerkz, all the sensors must be implemented in Java. Also the JPetStore application we used was written and built in 2003, when J2SE 5.0 was not available. Thus our test bed is built on J2SE 1.4, which means a lot of monitoring functions in J2SE 5 and 6 can not be used. However, there are alternative ways to set up the sensors for the test bed.

```
<!DOCTYPE aspectwerkz PUBLIC
    "-//AspectWerkz//DTD//EN"
    "http://aspectwerkz.codehaus.org/dtd/aspectwerkz.dtd">

<aspectwerkz>
    <system id="webapp">
        <aspect class="com.infosys.setlabs.j2ee.performance.
            instrumentation.InstrumentJavaAdvice">
        <pointcut name="allPublic" expression="execution(public
             * *..*.*(..)) AND within(com.ibatis.jpetstore..*)
            "/>
             <advice name="trace" type="around" bind-to="
                allPublic"/>
        </aspect>
    </system>
</aspectwerkz>
```

Figure 4.1: A sample Aspectwerkz deployment description file

4.2.1 CPU Usage

CPU usage is calculated over a short time interval, not for a single instant. It is the percentage of CPU time spent on processing computer programs within the total elapsed real time, which includes both the CPU busy time and CPU idle time. A typical modern CPU may work in three modes: user mode, kernel mode, and idle mode. To calculate the CPU usage from time A to time B, we first need to know the total time CPU has spent on all these three modes at time A and B. Then CPU usage from A to B is calculated as Formula 4.1. Here UT refers to time spent in user mode, KT refers to time spent in kernel mode, and IT refers to time spent in idle mode. If the CPU usage of a specific process is wanted, then we need to know the CPU time spent for that process in user mode and kernel mode.

$$CPU\_usage = \frac{(UT(B) + KT(B)) - (UT(A) + KT(A))}{(UT(B) + KT(B) + IT(B)) - (UT(A) + KT(A) + IT(A))} \tag{4.1}$$

Now the problem is how to get the CPU time spent in user mode, kernel mode, and idle mode. Unfortunately, it is impossible to get this information using pure Java 1.4 because there is no API to provide such functions. Two alternative approaches can be used to solve this problem. The first approach is to use the exec()

method in the Runtime Java class to call an external, platform-specific command like wmic on the Windows OS, and parse its output to get the information we need. This approach takes about 500ms to execute the wmic command and parse the output, which is too slow for our needs. Thus we use the second alternative approach in our experiment: integrate a segment of C code that implements these functions into our Java project via Java Native Interface (JNI). The API, called JavaSysMon, which we introduced in Section 2.3.3 already implements this function, which runs in 1ms, a very low overhead. We use this API in our test bed for CPU usage instrumentation.

4.2.2 Memory Usage

In Java, it is easy to know the free memory using the method freeMemory() in the Runtime class. However, it is the amount of free memory in the Java virtual machine only. If we want to know the total free memory in the operating system, this approach is not viable. Here we use JavaSysMon again to get the information we need. This API also provides a function to obtain the process table with the memory in bytes each process takes.

4.2.3 I/O Rate

The I/O operation number can be recorded by the AOP technique. In our experiment, we use an additional Aspectwerkz class to monitor all the methods in the Java I/O class. Every time an I/O related method is executed, the number of I/O operations is increased. Then we use the total I/O operation number to calculate I/O rate as Formula 4.2. Here N_k and T_k refer to the total number of file I/O operations detected and the system time when pattern k ends respectively.

$$IO\_Rate_k = \frac{N_k - N_{k-1}}{T_k - T_{k-1}} \tag{4.2}$$

4.2.4 Request Rate

There are a lot of ways to capture the requests sent to the server from the client side. One way is to analyze network packets like Wireshark [12]. This approach is able

to cover every event happening on the network, but it is difficult to integrate with our instrumentation system. Another approach is to add a listener on each servlet of the web application, but this requires modifying all the related JSP and servlet source code. Here we use the Apache server access log to capture the information of received user requests for JPetStore. A sample fragment of access log file is shown in Figure 4.2. The items listed from left to right on each line are: IP address of the client which sends the request to the server, date the server recieved the request, the method used by the client, the resource requested by the client, the protocol used by the client, the status code that the server sends back to the client, and the size of the object returned to the client.

```
1 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/ HTTP/1.1" 200 1488
2 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/bkg-topbar.gif HTTP/1.1" 200 959
3 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/sign-in.gif HTTP/1.1" 200 257
4 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/help.gif HTTP/1.1" 200 134
5 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/cart.gif HTTP/1.1" 200 96
6 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/separator.gif HTTP/1.1" 200 46
7 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/search.gif HTTP/1.1" 200 323
8 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/logo-topbar.gif HTTP/1.1" 200 3808
9 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /
    JPetStoreApp/images/poweredby.gif HTTP/1.1" 200 2722
10 142.244.142.60 - - [14/Dec/2011:16:34:43 -0700] "GET /favicon.
    ico HTTP/1.1" 200 21630
11 142.244.142.60 - - [14/Dec/2011:16:34:47 -0700] "GET /
    JPetStoreApp/shop/index.shtml HTTP/1.1" 500 3854
```

Figure 4.2: A Sample Access Log Recorded by Tomcat

Every time a request is sent to the server, an additional record of information about this request is added to the access log in one line. We track this access log file and report a new request when a new line in the file is found. In this way the number of requests sent to the server is well monitored. Because we do not parse the log, this approach does not bring too much overhead. In this way the total number of requests received is acquired, and the request rate can be calculated by Formula 4.3.

Here R_k and T_k refer to the total number of requests received and the system time when pattern k ends respectively.

$$Request\_Rate_k = \frac{R_k - R_{k-1}}{T_k - T_{k-1}} \tag{4.3}$$

4.3 Performance Degradation Injection

Because of the lack of real-world data, we need to inject performance problems into the test bed manually in order to test our system. There are a lot of potential causes that could lead to a performance degradation, and in this thesis we simulate four common cases of performance problems: low CPU resource, low memory resource, high I/O rate, and high request rate.

To simulate the low CPU resource scenario, we use an open source tool called CPUKiller [4] which can slow down other processes by adjusting its CPU usage by any percentage. This tool runs a high-priority program which takes most of the CPU resource to slow down other applications running on the same machine. This tool can be used to generate a high CPU load for the application server machine. However, we need to manually set its CPU usage, which limits our ability to automate the injection process. Also the algorithm it uses to generate a high CPU load is not revealed. To inject low CPU resource problems to affect the web application directly, we insert the code of a complex graphic algorithm which is able to completely consume CPU usage.

For low memory resource and high I/O rate problems, there are a lot of ways to create a shortage of the resource over a short time. For the low memory resource problem, we use a short segment of zip compression code to consume the memory. The number of zipping loops can be used to adjust the duration of memory consumption, and this segment of code can be running either inside or outside the web application. For the high I/O rate problem, we use a segment of Java code which contains looping file I/O operations to simulate this scenario. Also the number of loops can be used to determine the I/O rate.

For the high request rate problem, we use AB (Apache Benchmark) [1] which

is an open-source software provided by Apache Software Foundation. This tool is originally designed for benchmarking the Apache HTTP (Hypertext Transfer Protocol) server, with statistics showing how your server performs when the client number increases. For example, when an Apache application is running on the server, we can use Apache Benchmark to send 1000 requests by 10 times using command "ab -n 1000 -c 100 http://142.244.142.54/JPetStoreApp/shop/index.shtml". Here option '-n' indicates the number of requests totally, while option '-c' indicates the number of requests to perform at a time. There are also some other options to control advanced testing, but in this thesis we use this example to simulate a high request rate scenario. After this command is executed, some information is displayed to help administrators to understand the capacity of users an application could serve, such as the time taken for this test, the transfer rate, and the percentage of the requests served within a certain time. However, in this thesis we only use this tool as a trigger of concurrent multiple user requests to JPetStore on the server from another machine.

There are many factors in the human operations from the client side that may affect the response time from the server. In order to test the improvement and overhead of our approach, we need to conduct our experiment in a repeatable way. In this project, we use a load generator called Sahi [11]. Sahi is an open source automation tool to test web applications, developed by Tyto Software. Sahi can record the user operations on a website into a script, and then replay the script repeatedly to reproduce the actions. Also a user can write his own script based on its scripting rules. There are two reasons we choose Sahi as our automation testing tool. First, it is written and built in Java, and can be easily integrated into our system. Secondly, Sahi is easy to get started with and provides all the functions we need to test our approach, such as script playback, looping, and timing. The sample script we used to test system overhead in this experiment is shown in Figure 4.3.

```
1  _click(_link("Enter the Store"));
2  for (var $i=0; $i<50; $i++){
3  _click(_image("sign-in.gif"));
4  _setValue(_textbox("username"), "admin3");
5  _setValue(_textbox("username"), "admin3");
6  _setValue(_textbox("username"), "admin4");
7  _setValue(_password("password"), "adminadmin");
8  _click(_imageSubmitButton("button_submit.gif"));
9  _click(_image("fish_icon.gif"));
10 _click(_link("FI-SW-01"));
11 _click(_link("EST-2"));
12 _click(_image("button_add_to_cart.gif"));
13 _click(_link("Golden Retriever"));
14 _click(_link("EST-28"));
15 _click(_image("button_add_to_cart.gif"));
16 _setValue(_textbox("EST-28"), "3");
17 _click(_imageSubmitButton("update"));
18 _setValue(_textbox("keyword"), "poodle");
19 _click(_imageSubmitButton("search.gif"));
20 _click(_link("Cute dog from France"));
21 _click(_link("EST-8"));
22 _click(_image("button_add_to_cart.gif"));
23 _click(_image("button_remove.gif"));
24 _click(_image("button_checkout.gif"));
25 _click(_image("button_continue.gif"));
26 _setValue(_textbox("order.creditCard"), "1234 5678 8765 4321");
27 _click(_imageSubmitButton("button_submit.gif"));
28 _click(_image("button_continue.gif"));
29 _click(_image("my_account.gif"));
30 _click(_image("sign-out.gif"));
31 }
```

Figure 4.3: The Sahi Script Used to Test System Overhead

4.4 Analysis of Results

In order to evaluate our approach against an existing approach used by Glassbox on performance degradation detection, we conduct an experiment on the JPetStore web application. As described in Section 3, our approach needs an offline training step. So at first, we write 10 different Sahi scripts and run these scripts on an instrumented JPetStore application normally, where no performance problems are injected. Each script is executed 5 times, resulting in around 75,000 method execution records with over 255 different methods in total. Then we select around the 6,000 most frequent patterns into a pattern repository by our pattern mining algorithm as described in Chapter 3. For each pattern we calculate a Gaussian model based on the pattern

execution time. In this experiment, only patterns that occur more than 50 times are selected for the pattern repository.

We use four scripts to test our detection algorithm, with each script corresponding to one type of performance problem. Performance problems are injected at random times when the application is running. Each time a method returns, our system scans the pattern repository to see if there is a matched pattern. If yes, then we use the trained pattern model to calculate the probability that a performance degradation has happened. To simulate the simple detection method used by Glassbox, we alert whenever the execution time of one method exceeds a predefined threshold of 500ms.

To test the diagnosis phase, we use the same four scripts from the detection phase. However, for low CPU resource, low memory resource, and high I/O rate problems, we make additional runs on a recompiled web application version with faults injected at certain methods. Each time the faulty method is executed, the injected problem will be triggered.

4.4.1 Time Overhead

Our performance monitoring tool works continuously at runtime, which means the instrumentation, detection and diagnosis parts of our system are all attached to the web application. Whenever a method is executed, information is collected for its execution time, and more system information is consulted during diagnosis. Thus, we evaluate the time overhead of our performance monitoring system to make sure it does not slow down the web application greatly.

The metric we used to evaluate the overhead is the running time of the web application on the client side. In this experiment we repeatedly run a Sahi script to simulate a shopping scenario, in order to accurately calculate the time it takes to finish. At first, we run the script on the pure application without the instrumentation or detection system. Next, we activate our instrumentation system to test the overhead brought by each sensor. Finally, we activate our online detection and diagnosis system to test the total overhead of our tool. The Sahi script we use is the one in Figure 4.3, and the result is recorded in Table 4.1.

Table 4.1: System Overhead

	1 turn(s)	5 turns(s)	20 turns(s)	50 turns(s)
Pure application	18	55	223	596
Instrument on CPU	18	53	217	603
Instrument on memory	21	74	303	787
Instrument on I/O rate	17	55	210	600
Instrument on request rate	17	54	212	580
Full Instrumentation	21	68	264	725
Complete system	22	70	280	723

From this table, we see the overhead brought by our system is not too much. Even when all the functions are running in real-time, the execution time of the Sahi script is only increased by 21% based on the 50 turns data. Also we find that the major additional overhead is due to the memory sensor only. The overhead brought by detection and diagnosis is insignificant. Thus it is feasible to apply this system on commercial web applications. However, due to the limitations in this experiment, it is still not tested if the response time will have more overhead when there are a large number of users connected simultaneously, which is a very common case in most commercial web applications.

4.4.2 Performance Degradation Detection

As we described before, our performance degradation detection system works based on the execution time of each method. Thus it is important to know whether the execution time of each method is stable. That is, a method's execution time does not vary greatly depending on the system and time under normal situations. In the methodology, we assume that the execution time of a method call fits in a normal distribution without large variance. In this experiment, We run the JPetStore application under different system conditions at different times, and collect the execution time of each method to see whether the execution time varies greatly. For example, consider a method from all the 255 methods, occurring at three different times of day for 624 calls in total, with a distribution histogram of execution times as shown in Figure 4.4. We can see the execution time of this method is basically a nor-

mal distribution, which supports our assumption in the methodology part. We also checked other methods and about 90% of them fit a normal distribution in a normality test with significance level 0.1. This result makes our assumption on execution time more valid.

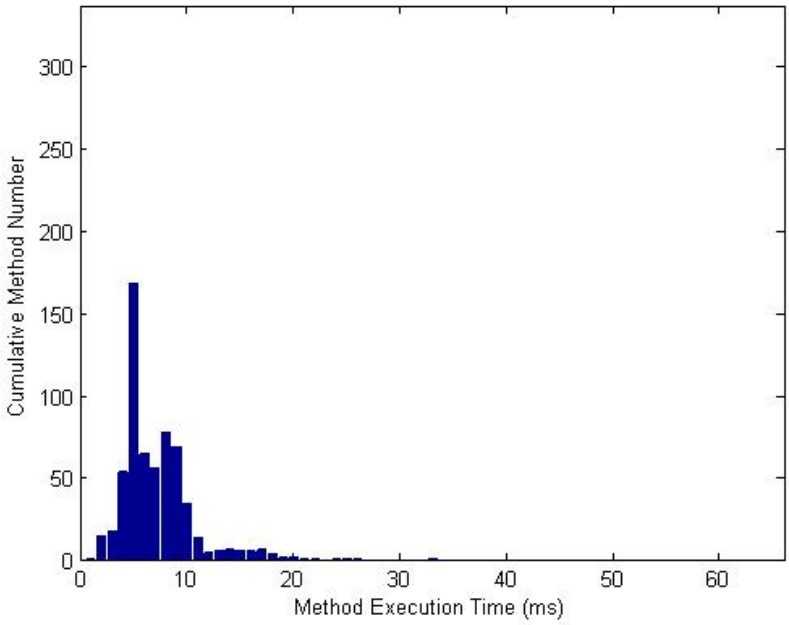

Figure 4.4: Distribution of a Method Call Execution Time

Next, we evaluate the detection phase of our approach. First we give an intuitive example comparing our approach against a fixed time threshold. We inject two performance problems (a low CPU resource problem and a high request rate problem) into the web application, and the time when performance degradation happens and ends are given in Figure 4.5(a). Also we plot the execution time of each method in Figure 4.5(a). Consecutive method calls appear on the horizonal axis. When the web application is running normally, the execution time of some methods can be extremely high, like the first few methods in the method call trace which even exceed 4000ms. At the same time, when the degradation happens, some methods can still be finished very fast. This tricky situation makes a single time threshold approach unable to detect performance degradation accurately. For 306ms, the single threshold leading to highest precision, we get the detection result in Figure 4.5(b).

There are a couple of false positives and many false negatives.

However, the dividing line between slow server and normally running server can be very clear by our approach, which is based on the Gaussian model of each method or pattern rather than a simple threshold. In Figure 4.5(c) we give the calculated probability that a method is running in a normal situation. We can see when performance degradation happens, there is an obvious decrease in the probability. The detection result of our approach is shown in Figure 4.5(d). Here we present an alert of performance degradation when the probability is below 0.1. Compared to the time threshold approach, our probability-based approach is more accurate, with both high precision and recall.

In order to validate our result in terms of precision and recall, we extend this experiment to all the four performance problem types we consider. Because our framework detects and diagnoses performance degradations using information at the method call level, the precision and recall are evaluated based on the number of method calls. For example, if a method finishes under a low performance condition and is detected by our approach, this is called a true positive instance. On the contrary, if a method finishes under normal conditions and our approach claims it is running normally, this is called a true negative instance. In Table 4.2 we compare our approach against a time threshold based approach on all four types of performance degradation in terms of precision and recall.

Table 4.2: Our Approach Against Traditional Approach on Performance Degradation Detection

	Time Threshold Approach		Our Approach	
	Precision	Recall	Precision	Recall
low CPU resource	56.2%	21.0%	76.3%	67.2%
low memory resource	31.9%	34.2%	84.8%	83.7%
high I/O rate	91.5%	94.7%	95.7%	98.2%
high request rate	47.8%	10.8%	80.5%	89.3%

From this table we can see our approach outperforms the single time threshold approach on every injected problem, especially in terms of recall. For the high request rate problem we injected, the recall of the single time threshold approach

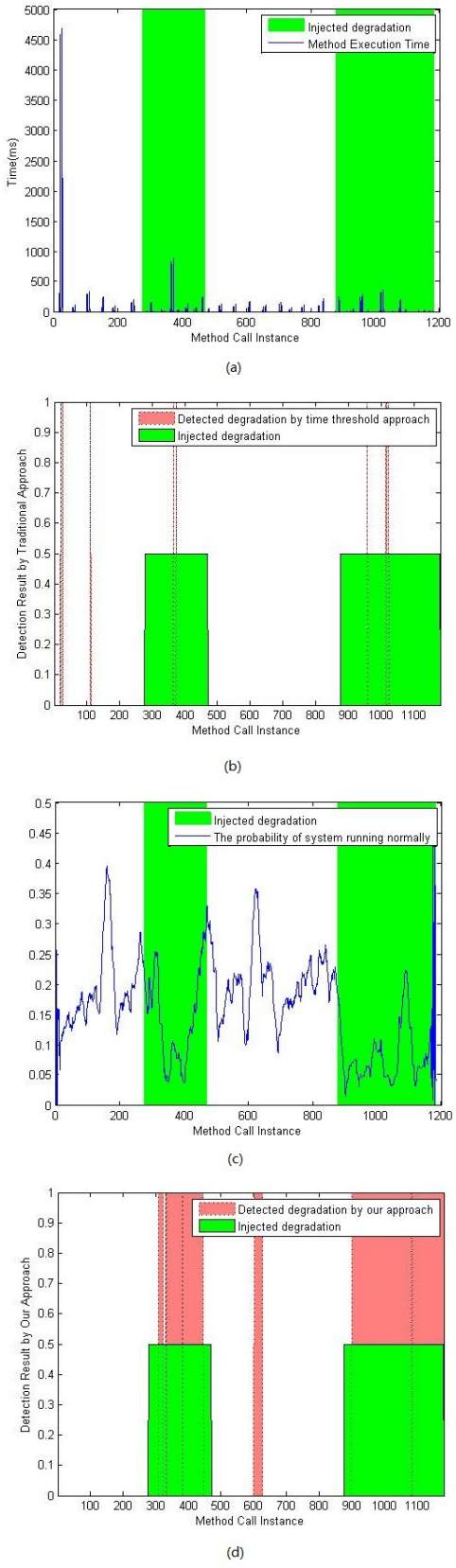

Figure 4.5: An Intuitive Performance Degradation Detection Experiment Result

is only 10.8%. The reason is obvious: when the web application performance is degraded, in most cases the execution time of each method call is increased proportionally rather than increased by a certain amount of time. For example, when 10 simultaneous requests are sent to the web application and all need the same method to process, the response time of this method might increase from 5ms to 50ms. This case is defined as a performance degradation, but the execution time of this method still does not reach the threshold set by administrator, typically 300ms. As a result, a large number of performance degradation cases are missed by the single time threshold approach, which leads to the low recall shown in Table 4.2. Similar things happen for the low CPU resource and low memory resource cases. The only exception is high I/O rate problem. Both the time threshold approach and our approach have high precision and recall. It is because when disk is full, the execution time of each method increases sharply. We find that it takes more than 500ms to finish for almost every single method when massive I/O operations happen. Thus it is a easy case for both approaches to detect accurately. On the other hand, for our approach we find that the result of low CPU resource detection is slightly worse than the other three cases. This result somehow indicates that the method execution time increment caused by low CPU resource may not be as significant as the other three problems.

4.4.3 Performance Degradation Diagnosis

In this part, we test our performance degradation diagnosis system. First, we evaluate the accuracy of categorizing all the four causes of performance problems we inject in this experiment. Next we test our approach on locating the injected method in the source code that best suggest the problem. We set $k = 20$ which means 20 previous method calls are suspected together with the current method call when the degradation was noticed. After the performance degradation is reported for 5 times, we collect the suspicious method calls involved and rank them by frequency. The results are shown in Table 4.3.

From this table we find the overall performance of our approach is effective on categorizing the cause, all the four injected problems can be classified with accuracy

Table 4.3: Performance Degradation Diagnosis Results

	Categorizing Cause	Locating Method
Low CPU resource	90.0%	Rank 1(5 methods)
Low Memory resource	82.8%	Rank 1(8 methods)
High I/O rate	87.5%	Rank 1(1 method)
High request rate	100%	N/A

over 80%. Specifically, all 27 methods with injected high request rate problems are correctly classified. This result should be attributed to the obvious increment on the number of requests when using Apache Benchmark for simulating simultaneous request traffic. In the method location part, we find that for all the three cases, the injected method is identified as the most suspicious. For low CPU and low memory resource problems, there are other methods equally suspicious, which means the administrator needs to check all these methods to find where the problem might lie. In the high I/O rate case, the injected faulty method is precisely ordered as the only suspicious method. We do not currently inject a high request rate problem into the methods of the web application, so the result about method location for this type of problem is not available.

4.5 Limitations and Threats to Validity

Some limitations of this work are:

1. Currently our sensors are still not enough to cover all the potential performance degradation causes. The sensors are all built based on Windows using Java, and it remains for future work to validate how they work when we apply our approach on web applications in other languages and platforms.

2. In performance degradation detection phase, we use method execution time as the only criteria. We believe the result can be further improved if some other sensor data can also be applied.

3. Our method location algorithm in diagnosis requires performance degradations to recur. The more times the degradation happens, the better our result

will be. Thus it becomes another question about how to locate the method when performance degradation first appears.

4. We lack the statistical study on the distribution of execution time over different methods. Execution time of certain methods such as database query operations may not fit the Gaussian distribution.

5. Our test bed is built on a sample pet store application, which is compiled using Java 1.4. How our approach works on a modern real world enterprise web application is still not validated.

Threats to the validity of the experimental results are as follows:

1. Due to the limitation of the number of clients, we can not simulate a real world enterprise web application, which usually has a lot of simultaneous clients connected. Also our computer, which runs the web application, is not as powerful as a commercial server. Thus the performance measurements might be different from the real world.

2. In the experiment we decide the value of certain parameters. The effect of these values on the results is not studied yet. It remains a question about how to decide the optimal parameters for detection and diagnosis.

3. Our injected problems in the evaluation are possibly too direct and easy to observe. Also, some complicated scenarios with multiple causes are not simulated in our experiment. It is still not studied whether our approach is robust when there are multiple potential problems. Also the interactions between different causes are not studied.

4.6 Summary

In this part we build a test bed to evaluate our approach from three aspects. At first, we find that the overhead of our system is acceptable, increasing response time by only around 21%. In the performance degradation detection phase, compared to the

sample approach Glassbox uses, our approach improves significantly in both precision and recall. For diagnosis, our approach performs well again. All the causes can be classified with accuracy over 80%. Also our algorithm can successfully locate the injected faulty method that causes recurrent performance degradations. All the results show that our system is capable of detecting and diagnosing web application performance degradations at the method call level with high accuracy, in real time.

Chapter 5

Related Work

There are a number of web application monitoring software products, such as Glassbox, CA Wily, SharePath, and HP Performance Center. These tools are commonly used in industry and are effective on performance monitoring. However, most only provide a dashboard for the administrators to analyze the data collected by the software, which means they do not detect and diagnose performance degradation by themselves. Glassbox is the only one which provides a way to detect performance degradation, but the approach is too simple. For cause diagnosis, they only create different charts about performance data and leave it for the administrator to do further analysis.

Java Application Monitor (JAMon) is a Java API for monitoring production application performance, developed by Steve Souza [8]. This tool is currently the most closely related work with our system. This tool collects performance statistics such as page hits, page execution time, simultaneous request numbers and so on. However, these statistics are not collected at the method call level, and it does not provide a performance degradation detection function either. This tool proposes a performance tuning function, but it is still up to the administrators to use the performance measurements provided by JAMon.

Some researchers also paid attention to web application performance monitoring, but none have a complete architecture including instrumentation, detection, and diagnosis. Nevertheless our work has overlap in certain aspects. In the web application instrumentation area, M. Schmid et al. [26] presented an instrumentation system which focused on multi-tier applications, and realized it on Apache Tomcat

and JBoss. Their instrumentation system is built on the ARM (Application Response Measurement) standard, and it focuses on method execution time too. They also tested their instrumentation system on JPetStore, and the overhead is 14.5% for a single client without capturing any other sensor values. N. Repp et al. [24] also presented a web service performance monitoring system, but it is based on the analysis of TCP/IP and HTTP protocols and throughput.

We also find previous work in the method call trace segmentation area [15, 27, 25, 30, 17]. Their goal is to extract some scenarios from the method trace, making it easier for the administrator to understand. Some machine learning techniques including K-means clustering techniques [27] or concept lattice analysis [25] are applied in their approaches. The results are encouraging, but their algorithms are applied on logs offline and are unable to do segmentation online in real time.

On web application performance diagnosis, S. Iwata et al. [18] proposed an approach for determining suspicious components in web applications when performance is degraded. Their approach is similar to our suspicious method location part, but uses method execution time as the measurement only. Besides, their work studies logs and is unable to analyze method call traces in real time. Other researchers also proposed different systems to diagnose performance related problems [21, 19, 23, 29], but unfortunately, all of them use the response time from server as the only measurement, without considering other parameters like CPU usage or memory usage of the server.

Chapter 6

Conclusion and Future Work

In this thesis, we design and implement a system to detect and diagnose web application performance degradation at the method call level in real time. We first implement an instrumentation system which monitors the web application from different aspects using Aspectwerkz and JavaSysMon. When the web application starts, the detection part of our system monitors and gives an alert when a performance degradation is found. Next the diagnosis part of our system suggests the type of cause, and makes a decision about whether this cause is inside the web application or not. If yes, our approach suggests the methods in the source code that reveal this performance degradation, assisting administrators to correct the problem.

Compared to other performance monitoring tools like Glassbox and CA Wily, our system does not only capture the server information when the web application is running, but also makes further analysis on the measurements collected. We build a test bed to evaluate our system and the results are encouraging. Precision and recall of detecting performance degradations reach over 80% for injected low memory resource, high I/O rate, and high request rate problems. Our results also outperform the single time threshold approach significantly. In diagnosis, our approach is effective, with accuracy over 80% for categorizing the four problem types injected. Moreover, the problematic method in source code can be located by our algorithm after the problem has reoccurred 5 times.

However, our approach can still be improved in future work. First, we need more sensors to test our detection and diagnosis algorithm over other causes, but the sensors should not generate high overhead. Secondly, because our evaluation

is conducted on a test bed which is built on a simple demo Java application, our approach is still not validated on real world enterprise web applications. In future work, we plan to test our system on a more complex web application with more components. Also the number of parallel clients should be increased to mimic a real world scenario more closely. Last but not least, our current algorithm is still too simple to analyze some complex scenarios, like performance degradation triggered by multiple interacting causes. Also some parameters and threshold values in our algorithm are decided by hand. Our approach could be more robust if some self-adaptive algorithms can be used in the future to determine these values.

Bibliography

[1] Apache Benchmark. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] Aspectwerkz. http://aspectwerkz.codehaus.org/.

[3] CA Wily Application Performance Management. http://www.ca.com/us/application-performance-management.aspx.

[4] CPU Killer. http://www.cpukiller.com/.

[5] Glassbox. http://glassbox.sourceforge.net/glassbox/Home.html.

[6] HP Performance Center. http://www8.hp.com/us/en/software/software-product.html?compURI=tcm:245-937011.

[7] iBatis JPetStore. http://sourceforge.net/projects/ibatisjpetstore/.

[8] Java Application Performance Monitor. http://jamonapi.sourceforge.net/.

[9] System Monitor API written in JAVA. https://github.com/jezhumble/javasysmon.

[10] The Apache Derby Project. http://db.apache.org/derby/.

[11] Web Automation and Test Tool. http://sahi.co.in/w/.

[12] Wire Shark. http://www.wireshark.org/.

[13] B. Basham, K. Sierra, and B. Bates. *Head First Servlets and JSP, 2nd Edition*. O'Reilly Media, 2008.

[14] J. Brittain and I.F. Darwin. *Tomcat: The Definitive Guide*. O'Reilly Media, Inc., 2007.

[15] F. Cantal de Sousa, N.C. Mendonca, S. Uchitel, and J. Kramer. Detecting Implied Scenarios from Execution Traces. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference*, pages 50–59, 2007.

[16] M. Crochemore. An Optimal Algorithm for Computing the Repetitions in a Word. In *Inf. Process. Lett.*, pages 244–250, 1981.

[17] J.L. Hellerstein, S. Ma, and C.S. Perng. Discovering Actionable Patterns in Event Data. *IBM Systems Journal*, 41(3):475–493, 2010.

[18] S. Iwata and K. Kono. Clustering Performance Anomalies in Web Applications Based on Root Causes. In *ICAC '11 Proceedings of the 8th ACM international conference on Autonomic computing*, pages 221–224, 2011.

[19] S. Jakobsson. Timing Failures Caused by Resource Starvation in Virtual Machines. In *DEPEND 2011, The Fourth International Conference on Dependability*, pages 88–91, 2011.

[20] M. W. Johnson. Monitoring and Diagnosing Applications with ARM 4.0 . In *In International Computer Measurement Group Conference*, pages 473–484, 2005.

[21] Parallel Data Laboratory. Diagnosing Performance Changes by Comparing System Behaviours. In *Supersedes Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-10-103*, 2010.

[22] R. Laddad. *AspectJ in Action, Second Edition*. Manning Publications, 2009.

[23] M. Raghavachari, D. Reimer, and R.D. Johnson. The Deployer's Problem: Configuring Application Servers for Performance and Reliability. In *Software Engineering, 2003. Proceedings. 25th International Conference*, pages 484–489, 2003.

[24] N. Repp, R. Berbner, O. Heckmann, R. Steinmetz, and Technische University Darmstadt. A cross-layer approach to performance monitoring of web services. In *ECOWS 2006 Workshop on Emerging Web Services Technology, IEEE*, 2006.

[25] K. Sartipi and H. Safyallah. Dynamic Knowledge Extraction from Software Systems using Sequential Pattern Mining. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 20(6):761–782, 2010.

[26] M. Schmid, M. Thoss, T. Termin, and R. Kroeger. A generic application-oriented performance instrumentation for multi-tier environments. In *In IEEE Intl. Symposium on Integrated Network Management*, pages 304–313, 2007.

[27] M. Smit, E. Stroulia, and K. Wong. Use Case Redocumentation from GUI Event Traces. In *Proceeding CSMR '08 Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 263–268, 2008.

[28] Tealeaf technology Inc. Open for Business? *Tealeaf technology Inc Whitepaper*, 2003.

[29] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. Automated Known Problem Diagnosis with Event Traces. In *EuroSys '06 Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 375–388, 2006.

[30] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process. In *In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR*, pages 134–142. IEEE Computer Society, 2005.