Delft University of Technology
Software Engineering Research Group
Technical Report Series

System Load Characterization Using Low-Level Performance Measurements

Cor-Paul Bezemer, Andy Zaidman

Report TUD-SERG-2012-006

TUD-SERG-2012-006

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

© copyright 2012, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

System Load Characterization Using Low-Level Performance Measurements

Cor-Paul Bezemer
Delft University of Technology
The Netherlands
c.bezemer@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

***Abstract*—The performance of a software system directly influences customer satisfaction. Self-adaptiveness can contribute to this customer satisfaction by (1) taking appropriate measures when the performance becomes critical, e.g., the system load is too high, or (2) scheduling intensive tasks when the load is low. We investigate how self-adaptive systems can use low-level system measurements to characterize the load on a system. Our approach uses a combination of statistics and association rule learning to perform the characterization. We evaluate our approach using two case studies: a large-scale industrial system and a widely used synthetic benchmark (RUBiS). From our case studies follows that our approach is capable of closely characterizing the load on a system and that it is successful in detecting performance anomalies as well.**

I. Introduction

An important property of self-adaptive systems is the ability to self-optimize [1]. This can, for example, be done by changing configuration parameters or adding resources, based on the analysis of monitored information. An important research challenge in the area of self-optimization is correlating low-level system measurements with high-level service objectives (SLOs) [1].

One of these high-level service objectives is application performance, as it directly influences customer satisfaction and therefore the success of an application [2]. Application performance is tightly coupled to the load on the system: when the system load becomes too high, the application performance will get worse [3]. By implementing techniques on a self-adaptive system which allow us to characterize the load on the system, we can improve the application performance and therefore the customer satisfaction by taking appropriate actions when necessary. In addition, load characterization gives users (e.g., system administrators) insight on when to perform intensive administrative tasks, such that they do not negatively affect the customer satisfaction.

While end-user performance is important for a wide range of software systems, it becomes all the more important when more users start to use the same system simultaneously. Examples of such a system are multi-tenant Software-as-a-Service (SaaS) systems, in which large groups of users are working with the same base-application. Performance has been explicitly defined as a challenge in this context due to the high number of users using the same resources [4].

It is our goal to characterize the system load on self-adaptive systems, i.e., to "sense" [5] changes in system load and to decide on which actions to take to improve the performance of the system.

Our approach is based on measuring a wide variety of low-level system measurements, so-called *performance counters* [6], such as the `Memory\Available Mbytes` and `Processor\%Processor Time` counters. We propose to use a combination of statistics and association rule learning to characterize the server load. We use association rule learning because association rules are human-readable and can give a first problem diagnosis. Our approach is evaluated in two case studies: (1) an industrial multi-tenant SaaS application and (2) a widely-used benchmark.

The outline of this paper is as follows. In Section II, we will motivate the problem and present our research questions. Section III discusses the metric we will use for our approach. In Section IV we present our approach for server load characterization using low-level system measurements. Our approach is evaluated in two case studies, which are presented in Section V to Section VII and discussed in Section VIII. We conclude our paper with a discussion of related work in Section IX and future work in Section X.

II. Problem Statement

In this paper, we focus on load characterization through low-level system measurements with the goal of improving the system performance and therefore the customer satisfaction. The goal of our approach is to characterize the performance of a system using such low-level measurements only, as these measurements are relatively cheap to make and easily accessible. The main research question addressed in this paper is:

> *How can we characterize the load on a system, based on low-level performance measurements?*

The challenges of automatically creating such a load characterization are discussed in the following paragraphs.

A. Selecting a Metric

Different applications have different performance requirements. For example, the required minimum throughput may be much higher on a news website than on a website which generates financial reports of a user's administration. In addition, report generation may take longer for a large company than for a small company. The performance of a system is closely connected to its load, as a system with high load is more likely to perform worse than a system with low

load. Therefore, we can pass judgement on the performance of a system by analyzing the system load. It is necessary to find a metric which characterizes system load and takes the differences between applications and (groups of) users (or tenants) into account. Using such a metric will let us validate performance requirements more precisely than using a simple metric such as average response time or throughput.

> **RQ 1.** *Which metric characterizes system load and takes differences in performance requirements between applications and (groups of) users into account?*

B. Relating State to Low-Level System Measurements

After deciding on the used performance metric, we must find a way of translating low-level system measurements to this metric. Our measurements consist of a wide variety of performance counters. Cohen et al. [7] have shown that combinations of low-level system measurements can give a better description of the state of a system than simple rules of thumb, which means an approach for correlating these measurements with a metric should take this into account.

> **RQ 2.** *How can we relate (combinations of) low-level system measurements to this performance metric?*

C. Detecting Performance Anomalies

Finally, the approach must be able to detect performance anomalies using these low-level system measurements. In order to prevent the self-adaptive system from taking unnecessary actions, it is important that the approach can distinguish isolated extreme measurements from longer-term performance problems.

> **RQ 3.** *How can performance anomalies be detected using these low-level system measurements?*

In the next section we discuss the metric that will be used in our approach. In the rest of this paper we propose an automatable approach for the characterization of system load using low-level measurements.

III. Selecting a Metric

Application performance can be expressed in many different metrics. One of the most important is average response time [3], as it strongly influences the user-perceived performance of a system. While a generic performance metric like average response time can give an overall impression of system performance, it does not make a distinction between different actions and/or users and may therefore exclude details about the performance state of a system. An example of this can be seen in a bookkeeping system: report generation will take longer for a company with 1000 employees than for a company with 2 employees. When using average response time as threshold setting for this action the threshold will either be too high for the smaller company or too low for the larger company. In addition, we want to be flexible in selecting an interval for our performance measurements. A metric such as average response time works over a longer period only, as it is relatively heavily influenced by batch actions with high response times (such as report generating scripts) when using short intervals. Therefore, we are looking for a metric which is (1) resilient to differences between users and actions and (2) independent of time interval length.

In order to define a metric which fits into this description, we propose to classify all actions as *slow* or *normal*. To decide whether an action was *slow*, we calculate the mean μ_{au} and standard deviation σ_{au} of the response time of an action a for each user u over a period of time. Whenever the response time rt_i of action a of user u is larger than $\mu_{au} + \sigma_{au}$, it is marked as *slow*, or:

For every action a_i and user u,

$$a_i \in \begin{cases} SLOW & \text{if } rt_i > \mu_{au} + \sigma_{au} \\ NORMAL & \text{otherwise} \end{cases}$$

Because μ_{au} and σ_{au} are calculated per action and user, this metric is resilient to differences between actions and users. Note that by doing this, we assume that the system has been running relatively stable, by which we mean that no significant long-lasting performance anomalies have occurred over that period of time. Another assumption we make is that an action will have approximately the same response time when executed by the same user at different times.

From this classification, we construct a metric for performance characterization which fits into our description, namely the ratio $SAratio_t$ (*Slow-to-All-actions-ratio*) of the number of slow actions $SLOW_t$ to the total number of actions in time interval t:

$$SAratio_t = \frac{|SLOW_t|}{|SLOW_t| + |NORMAL_t|}$$

Because it is a ratio, isolated extreme values have a smaller influence on the metric, which makes it independent of time interval[1]. However, it is important not to make the analysis interval too small (e.g., less than 30 seconds) to keep the overhead of making performance measurements low.

Now that we have decided upon the metric to use for our approach, the next challenge is to relate this metric to low-level system measurements. In the next section we will present our approach for doing this.

IV. Characterizing System Load Using Low-Level Measurements

The goal of our approach is to translate low-level system measurements into a classification of the load on a system. In this paper, we propose to use a combination of statistics and association rule learning to perform this translation.

[1]Unless the total number of actions is very low, but we assume this is not the case in modern systems.

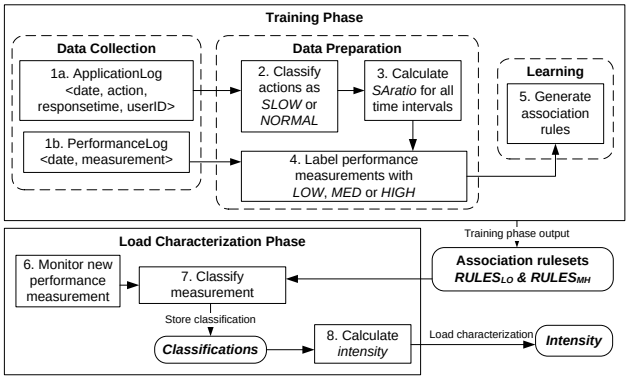


Figure 1. Steps of our approach for characterizing performance using low-level system measurements

Figure 1 depicts the steps of our approach, which are explained in more detail in this section. Our approach consists of two phases, the training phase and the load characterization phase. During the training phase, we collect and analyze logged actions and performance data of a system. After marking the monitored actions as *slow* or *normal* as described in the previous section, we calculate the $SAratio$ per time interval and use this to classify the system load during these intervals into three classes. From this data we learn association rules, which help us to decide whether the load on the system is *low*, *medium* or (too) *high* based upon low-level measurements during the load characterization phase. In our approach we use association rule learning because association rules can give us an indication of why a measurement received a certain classification, while other learning methods such as pattern recognition do not give this indication [8]. In addition, to detect performance issues during the load characterization phase we analyze the *intensity* metric using a sliding window approach.

We will now discuss the steps taken in our approach.

A. Training Phase

1) Data Collection: As every system is different and has different performance requirements, we must first analyze it to understand its characteristics. Therefore, we log all actions in the system, including their duration and the ID of the user that made them, for a period of time (*step 1a* in Figure 1). In parallel, we make low-level system measurements at a defined interval t (*step 1b*). This results in the following log files:

- A log file `ApplicationLog` containing the `date`, `action`, `responseTime` and `userID` (if existent) for every action made to the application
- A log file `PerformanceLog` containing low-level system performance measurements and the date at which they were made

2) Data Preparation: After collecting the data, we classify all actions in the `ApplicationLog` as $slow$ or $normal$ (*step 2*) and calculate the $SAratio_t$ per time interval t as described in Section III (*step 3*).

From a performance point of view, we are interested in three types of observations for the value of $SAratio$:

- $high$: system load is typically too high (5%)
- med: system load may become or may just have been problematic (10%)
- low: system load is non-problematic (85%)

These *load classifications* form three classes $HIGH$, MED and LOW. As a threshold for the MED and $HIGH$ classes we use the 85^{th} and 95^{th} percentile of the distribution of $SAratio$. For an industrial system, i.e., a system from which we expect it to have relatively few performance problems, our expectation is that the $SAratio$ is distributed around the mean, following a normal or gamma-like distribution. By using the 85^{th} and 95^{th} percentile we use approximately the same confidence intervals as commonly used for the normal distribution [3]. We label all low-level measurements in the `PerformanceLog` with their corresponding load classification (*step 4*).

3) Learning: The final step of the training phase is to apply the association rule learning algorithm to the labeled data (*step 5*). From experimentation we know that association rule learning algorithms generate bad performing association rules for this type of data when trying to generate rules for the LOW, MED and $HIGH$ classes in one run. Therefore, we run the learning algorithm twice on different parts of the dataset to improve the classification. In order to prevent overfitting [9], we combine the MED and $HIGH$ classes into the temporary $OTHER$ class and we generate a random subset of the LOW class, that is approximately equal in size to the number of elements in the $OTHER$ class. We then run the rule learning algorithm twice:

- For separating the LOW and $OTHER$ classes $\rightarrow$ $RULES_{LO}$
- For separating the MED and $HIGH$ classes $\rightarrow$ $RULES_{MH}$

The final results of the training phase are the association rulesets $RULES_{LO}$ and $RULES_{MH}$. Table I shows some simplified sample input with a sample association rule.

Sample input			
SAratio	Server1\CPU idle	Server1\Mem used	Server2\Page in
0.26	60	25	83615
0.29	40	15	84615

Distribution SAratio: $85^{th} percentile = 0.21$, $95^{th} percentile = 0.27$
Sample association rule
(Server1\CPU idle $<$ 50 & Server2\Page in $>$ 83615) $\Rightarrow$ high

Table I
SAMPLE INPUT AND ASSOCIATION RULE

B. Load Characterization Phase

During the load characterization stage new, unlabeled low-level measurements are monitored (*step 6*) and classified into

one of the load classes LOW, MED and $HIGH$ using the rulesets. First, the measurement is classified into the LOW or $OTHER$ class using the $RULES_{LO}$ ruleset. When it is classified into the $OTHER$ class, it is classified again using the $RULES_{MH}$ ruleset to decide whether it belongs to the MED or $HIGH$ class (*step 7*).

As explained, measurements classified into the $HIGH$ class often indicate a performance problem. Therefore, we want to emphasize the occurence of adjacent measurements classified into this class, as this would help to distinguish isolated extreme measurements from a longer-term performance problem (*step 8*).

Algorithm 1 CALCINTENSITY$(n, clasfSet, intensity)$

Require: Window size n, a set of load classifications $clasfSet$, the current $intensity$.
Ensure: The $intensity$ of the last n classifications is added to the current $intensity$.

```
1: window = clasfSet.getLastItems(n)
2: cntLow = count(window, LOW)
3: cntMed = count(window, MED)
4: cntHigh = count(window, HIGH)
5: maxCnt = max(cntLow, cntMed, cntHigh)
6: if maxCnt == cntHigh then
7:       intensity = intensity + 2
8: else if maxCnt == cntMed then
9:       intensity = max(intensity − 1, 0)
10: else
11:      intensity = max(intensity − 2, 0)
12: end if
13: return intensity
```

To do this, we use a sliding window approach. A window of size n contains the load classifications of the last n measurements. We count the occurences of LOW, MED and $HIGH$ objects and keep a counter $intensity$. Everytime the majority of the objects in the window have a $HIGH$ classification, $intensity$ is increased by 2. When the system returns to normal performance, i.e., the majority of the load classifications in the window are MED or LOW, $intensity$ is decreased by 1 and 2 respectively. These steps are depicted by Algorithm 1 (CALCINTENSITY). Figure 2 shows the effects of this algorithm.

V. EXPERIMENTAL SETUP

A. Subject Systems

In order to evaluate our approach, we performed two case studies on SaaS systems: (1) on a real industrial SaaS application running on multiple servers and (2) on a widely-used benchmark application running on one server. Although the application used in our industrial case study is not yet self-adaptive, our approach forms the first step towards enabling it to take self-adaptive actions.

In the rest of this paper we will assume the `ApplicationLog` contains requests made to the application (i.e., the webserver log — records will have the format `date`, `page`, `responseTime`, `userID`). In this section, the experimental setup of the case studies is presented. In the next sections, the specific details and evaluation are discussed per case study.

B. Process

Training Phase: The `ApplicationLog` and `PerformanceLog` are collected using the webserver and OS-specific tools and are imported into a SQL database for the data preparation phase. All steps in the data preparation phase are performed using a sequence of SQL queries. The generation of the *MED*, *HIGH* and *OTHER* classes and the subset of the *LOW* class is done by custom implementation in Java. For the implementation of the rule learning algorithm we have used the JRip class of the WEKA API [10], which is an implementation of the RIPPERk algorithm [11]. We used the JRip algorithm because it is a commonly used association rule learning algorithm and experimentation showed that this algorithm gives the best results for our dataset with respect to classification error and speed.

Load Characterization Phase: The load characterization phase was implemented in Java, resulting in a tool that can be used on newly monitored data. The $intensity$ calculated by this tool is sent to XML output and an $intensity\ graph$ is generated using JFreeChart[2], similar to Figure 2(b).

C. Relation to Research Questions

In our case studies we will evaluate our answer to the following research questions:

> **Main RQ.** *How can we characterize the load on a system, based on low-level performance measurements?*
>
> **RQ 3.** *How can performance anomalies be detected using these low-level system measurements?*

At the end of each case study section, we will reflect on whether our answer to these research questions was satisfactory.

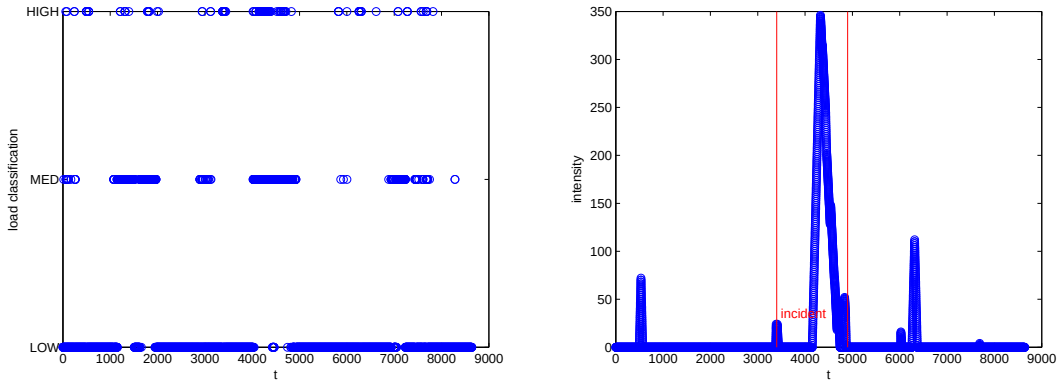
VI. CASE STUDY: EXACT ONLINE

Exact Online[3] (EOL) is an industrial multi-tenant SaaS application for online bookkeeping with approximately 18,000 users[4]. The application currently runs on several web, application and database servers. It is written in VB.NET and uses Microsoft SQL Server 2008.

A. Training Phase

1) Data Collection: Exact Online performance data is stored for a period of 64 days in the form of logged performance counter values. Table III depicts the subset of performance counters which are being logged. This list was selected by Exact performance experts and contains the

[2]http://www.jfree.org/jfreechart/
[3]http://www.exactonline.nl
[4]In fact, there are about 10,000 users with 18,000 administrations, but for clarity we assume 1 user has 1 administration throughout this paper.



(a) Before (b) After

Figure 2. Load classification of the EOL incident before and after applying CALCINTENSITY

performance counters most commonly used during performance analysis. Therefore, we limited our case study to the analysis of these performance counters recorded during 64 days. Table II shows some details about the collected data.

	Exact Online	RUBiS
`ApplicationLog`		
# actions	88900022	853769
# applications	1067	33
# users	17237	N\A
# (application, user)-tuples	813734	N\A
monitoring period	64 days	30 minutes
`PerformanceLog`		
# measurements	182916	1760
# performance counters	70	36
measurement interval	30s	1s

Table II
DETAILS ABOUT DATA COLLECTED DURING THE CASE STUDIES

The `ApplicationLog` was retrieved by selecting the required elements from the Internet Information Server log, which is stored in a SQL database. The performance measurements were logged into a database called `PerformanceLog` by a service which collects performance counter values at set intervals on all servers. These intervals were configured by company-experts, based on their experience with the stability of the counters.

2) Data Preparation: To prepare the data for association rule learning, a number of SQL queries were executed. To verify that the response times of each application are approximately normally distributed per user, we have inspected the histogram of 10 *(application, user)*-tuples which were ranked in the top 30 of tuples with the highest number of actions. The tuples were selected in such a way that there was a variety of users and applications. This inspection showed that the response times follow the lognormal distribution, which is consistent with the results found for think times (equivalent to response times) by Fuchs and Jackson [12]. Table IV displays the percentage of actions in the $NORMAL$ and $SLOW$ classes for each sample based on the logarithm of the response time. As shown in the table, the percentage of actions in the classes are close to what one would expect when assuming the (log)normal distribution.

Virtual Domain Controller 1 & 2, Staging Server	
Processor\%Processor Time	
Service 1 & 2	
Memory\Available Mbytes	Process\%Processor Time
Processor\%Processor Time	System\Processor Queue Length
SQL Cluster	
LogicalDisk\Avg. Disk Bytes/Read	LogicalDisk\Avg. Disk Read Queue Length
LogicalDisk\Avg. Disk sec/Read	LogicalDisk\Avg. Disk sec/Write
LogicalDisk\Avg. Disk Write Queue Length	LogicalDisk\Disk Reads/sec
LogicalDisk\Disk Writes/sec	LogicalDisk\Split IO/sec
Memory\Available Mbytes	Memory\Committed Bytes
Memory\Page Reads/sec	Memory\Pages\sec
Paging File\%Usage	Processor\%Processor Time
Buffer Manager\Lazy writes/sec	Buffer Manager\Buffer cache hit ratio
Buffer Manager\Page life expectancy	Databases\Transactions/sec
Latches\Average latch wait time (ms)	Latches\Latch Waits/sec
Locks\Lock Waits/sec	Memory Manager\Memory grants pending
General Statistics/User Connections	SQL Statistics\Batch requests/sec
SQL Statistics/SQL compilations/sec	virtual\vfs_avg_read_ms
virtual\vfs_avg_write_ms	virtual\Bytes_IOPS
virtual\vfs_io_stall_read_ms	virtual\vfs_io_stall_write_ms
virtual\vfs_NumBytesRead	virtual\vfs_NumBytesWritten
virtual\vfs_Num_IOPS	virtual\vfs_NumReads
virtual\vfs_NumWrites	
Webserver 1 & 2	
ASP.NET\Requests Current	ASP.NET\Requests Queued
ASP.NET Apps\Req. Bytes In Total	ASP.NET Apps\Req. Bytes Out Total
ASP.NET Apps\Req. in App Queue	ASP.NET Apps\Requests Total
ASP.NET Apps\Req./sec	Memory\Available Mbytes
Process\%Processor Time	Process\Handle Count
Process\Thread Count	Processor\%Processor Time

Table III
LIST OF MONITORED PERFORMANCE COUNTERS FOR EOL

The deviations are caused by the fact that these response times were monitored in a real environment, rather than a perfect environment without external influences [13].

Figure 3 shows the distribution of $SAratio$ in the EOL case study, together with the 85^{th} and 95^{th} percentile.

3) Learning: Running the association rule learning algorithm on the EOL dataset resulted in a ruleset $RULES_{LO}$ of 27 rules and a ruleset $RULES_{MH}$ of 29 rules.

B. Load Characterization Phase

To validate the rulesets for EOL, we have analyzed data which was monitored during a performance incident. During this incident a bug in a product update caused logfiles to be locked longer than necessary, which resulted in bad perfor-

Sample #	% NORMAL	% SLOW	# actions
EOL1	85.82	14.18	2736563
EOL2	89.64	10.36	1450835
EOL3	92.74	7.26	599470
EOL4	89.02	10.98	351494
EOL5	85.29	14.71	270268
EOL6	78.72	21.28	211481
EOL7	82.77	17.23	161594
EOL8	91.33	8.67	144050
EOL9	84.31	15.59	112867
EOL10	91.46	8.54	97793
RUBIS1	85.32	14.68	35651
RUBIS2	84.60	15.40	23262
RUBIS3	85.80	14.20	19842
normal distribution	84.2	15.8	

Table IV

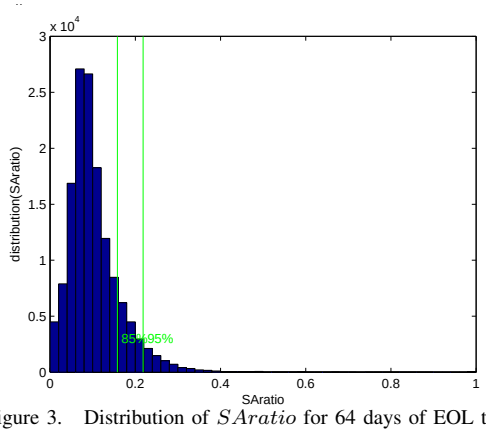


Figure 3. Distribution of $SAratio$ for 64 days of EOL traffic

mance. Note that the incident happened 3 months after the training data was recorded, which makes it a strong test case as the training data and incident data are not biased towards each other. To validate the rulesets, we have performed the load characterization phase steps as described in Figure 1. Note that this means we used only the `PerformanceLog` to characterize the system load during the incident. Figure 2(b) graphs the *intensity* calculated after classifying all measurements in the `PerformanceLog` of the 3 days surrounding the incident. The bug was introduced around $t = 3400$ and solved around $t = 4900$. Figure 2(b) shows a high peak from approximately $t = 4100$ to $t = 4900$, which indicates our approach is capable of detecting performance anomalies. Note that the performance anomaly was detected later than it was introduced because at the time of introduction there were very few users using the application which left the anomaly temporarily unexposed. The other, lower peaks were caused by heavier system load during administrative tasks such as database maintenance, which are performed at night for EOL.

C. Evaluation

Revisiting the main RQ, the EOL case study shows that the load can be characterized closely by the approach as we were able to identify 'normal' peaks and an incidental peak in the intensity graph easily, even for data which was monitored 3 months after the data with which the rulesets were trained.

Revisiting RQ 3, the EOL case study shows that the approach is capable of detecting performance anomalies, as the calculated intensity was much higher during the incident.

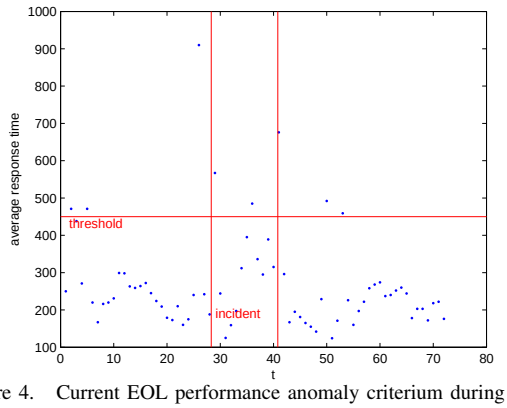


Figure 4. Current EOL performance anomaly criterium during incident

As a comparison, Figure 4 shows the performance anomaly criterium used by the EOL team. In this criterium, an anomaly is reported when the average response time exceeds 450 ms in an hour. Figure 4 shows that shortly after the start of the incident an anomaly was reported, however:

- This report was not handled until 4 hours later when working hours started.
- This report was not considered an anomaly because the average response time dropped to an acceptable value after the report, i.e., the report was considered an isolated measurement due to long-running administrative tasks.

At $t = 36$ another anomaly report was sent, which was investigated and lead to a solution around $t = 40$. However, this was also an isolated measurement which lead to confusion for the performance engineers.

Using our approach, the performance engineers would have had a stronger indication that a performance anomaly was occurring as it shows a continuous performance problem during the incident. In addition, our approach would have reported the anomaly between $t = 34$ and $t = 35$.

VII. Case Study: RUBiS

RUBiS [14] is an open source performance benchmark tool which exists of an auction site and a workload generator for this site. The auction site is written in PHP and uses MySQL as database server. The workload client is written in Java. We have installed the auction site on one Ubuntu server, which means that the web and database server are both on the same machine. The workload client was run from a different computer running Windows 7.

A. Training Phase

1) Data Collection: In order to generate several traffic bursts, we have configured 3 RUBiS workload clients to run for 30 minutes in total. Figure 5 shows the number of hits per second generated by the clients. The number of hits generated was chosen after experimentation to reach a level where the computer running the client reached an overloaded

state. The goal of this was to validate that our approach detects performance anomalies on the server only and is not affected by problems on the system of the client.

Table V shows the set of performance counters monitored on the server. The performance counters were monitored with DStat[5] every second and the output was redirected to a file *dstatlog*. After the benchmark completed, the Apache *access_log* and *dstatlog* were parsed into the SQL databases `ApplicationLog` and `PerformanceLog`. These databases had the same structure as those in the EOL case study so that the same queries could be used. Table II contains some statistics about the collected data.

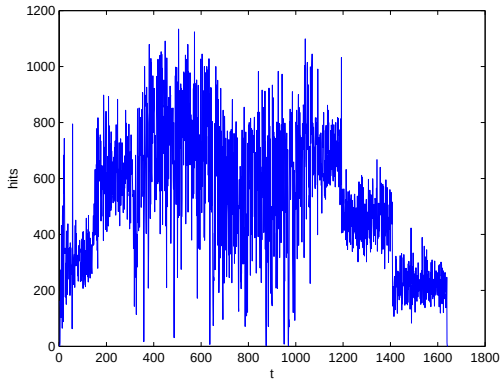


Figure 5. Traffic generated during the RUBiS case study

CPU stats	**Memory stats**
system, user, idle, wait	used, buffers, cache, free
hardware & software interrupt	**Process stats**
Paging stats	runnable, uninterruptable, new
page in, page out	**IO request stats**
Interrupt stats	read requests, write requests
45, 46, 47	asynchronous IO
System stats	**Swap stats**
interrupts, context switches	used, free
Filesystem stats	**File locks**
open files, inodes	posix, flock, read, write
IPC stats	
message queue, semaphores	
shared memory	

Table V
LIST OF MONITORED PERFORMANCE COUNTERS FOR RUBIS

2) Data Preparation: Because the applications in RUBiS perform equal actions for all users, we did not calculate the mean and standard deviation per (application, user)-tuple but per application instead. To verify that the response times of each application are approximately normally distributed, we have inspected the histogram of the 3 applications with the most hits. Table IV shows the number of *slow* and *normal* requests for these applications. Figure 6 shows the distribution of $SAratio$ for the RUBiS case study, together with the 85^{th} and 95^{th} percentile. Note that the distribution is different from the distribution of $SAratio$ in the EOL case study as the type of application and workload are different.

3) Learning: Performing the association rule learning algorithm resulted in a ruleset $RULES_{LO}$ of 6 rules and a ruleset $RULES_{MH}$ of 2 rules.

[5]http://dag.wieers.com/home-made/dstat/

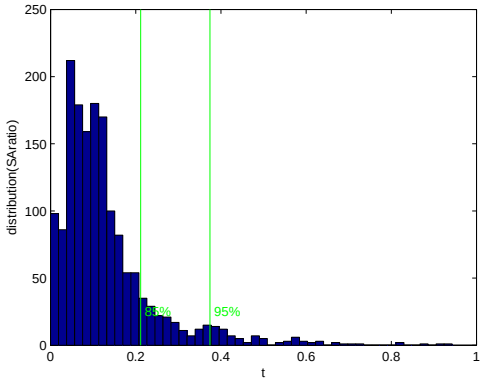


Figure 6. Distribution of SAratio for 30 minutes of RUBiS traffic

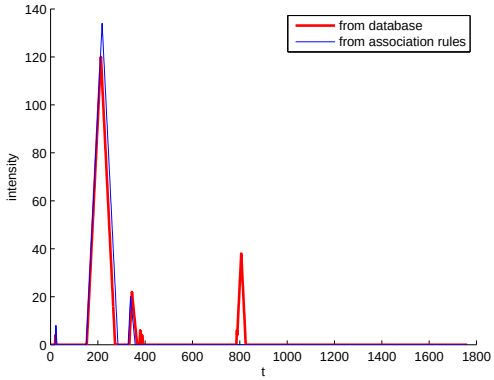


Figure 7. Intensity calculated from database and from assocation rules

4) Load Characterization: To validate our approach for the RUBiS case study we calculated the intensity directly from the `ApplicationLog` using the $SAratio$ and using assocation rules. Because the association rules were generated from a subset of the `PerformanceLog` as described in Section IV-A3, part of the data to classify was used as training data. We deliberately did this to include the data generated by the overloaded client in the classification. Nonetheless some new data was classified during the load characterization phase. Figure 7 shows the intensity graph of both calculations[6]. While the graphs are very similar, the calculation from the database shows a peak around $t = 800$, whereas the other does not. This can be explained by inspecting Figure 5 which shows a drop in the generation of traffic around $t = 800$. This happened because the client was overloaded at that time. Due to the implementation of RUBiS which uses synchronous connections [15], i.e., the client waits for a response from the server after sending a request, the response times went up[7]. Therefore, the calculation from the database, which uses the response times directly, decides that the server load is high. However, the association rules classify the monitored performance counter values as normal server load, which is correct. The peak around $t = 200$ can be explained by the fact that the workload client executes

[6]This graph is best viewed in color.

[7]Apache logs the time to *serve* the request, i.e., the time between receipt of the request by the server and receipt of the response by the client

certain heavy search queries for the first time. After this the results are cached, resulting in less load on the server.

B. Evaluation

Revisiting the main RQ, the RUBiS case study shows our approach is capable of characterizing the load on the server well, as demonstrated by Figure 7. In fact, our approach is more precise than the approach that relies on the average response time directly, as our approach did not classify the overloaded client as a server performance problem.

Revisiting RQ 3, in the RUBiS case study we did not reach the point of performance problems on the server. However, some weaknesses of the RUBiS implementation were exposed. These weaknesses, mostly caused by the database design, can be considered performance bugs [16]. In addition, we have shown that our approach is not affected by performance problems on the client.

VIII. Discussion

In this section we will map the research questions that we set out in Section II to the results we obtained from our experiment. Furthermore, we also identify threats to validity that might impact our conclusions.

A. The Research Questions Revisited

1) Revisiting Main RQ: Load Characterization: In our problem statement, we questioned how to characterize the load on a system using low-level performance measurements, such that the performance of the system can be improved. After our evaluation using two case studies, we can conclude that our approach (1) is able to characterize the actual system load closely and (2) can assist with detecting performance anomalies. We have shown this in a case study on an industrial application running on several servers and in a case study on an application with synthetically generated workload running on one server.

2) Revisiting RQ 1: Selecting a Metric: In our evaluation we have shown that the use of the $SAratio$ in combination with the $intensity$ offers a way of describing the performance of a system which takes into account differences between users and works well for short time intervals. This allows for more precise load characterization than a simple metric such as the average response time. Note that we make the assumption that the response time is approximately (log)normally distributed for a (application, user)-tuple, which allows us to use the mean and standard deviation to define $slow$ and $normal$ requests. For applications of which the response times do not follow such a distribution, this assumption cannot be made.

3) Revisiting RQ 2: Classifying Low-Level Measurements: In our case studies we have shown that our approach is capable of translating low-level performance measurements into a load classification successfully. For the definition of the LOW, MED and $HIGH$ classes we have used fixed percentiles based on the assumption that the $SAratio$ follows a normal or gamma-like distribution. We believe, that in the rare case when $SAratio$ does not follow such a distribution, the percentiles are still appropriate. A good example of this is the RUBiS case study, during which little extreme values of $SAratio$ occurred but in which the calculated $intensity$ still gave a good representation of the load. One sidenote on the RUBiS case study is that it was evaluated (partly) on the data used during the training phase. However, as we create a subset of the training data before we apply the association rule algorithm, we feel this should not strongly affect the evaluation.

4) Revisiting RQ 3: Detecting Performance Anomalies: In our EOL case study we have shown that our approach is capable of detecting performance anomalies. The anomaly was detected 3 months after the training data was recorded, which adds to the credibility of the approach as it is also capable of detecting new incidents, which did not occur in the training set. During this case study we noticed as well that our approach is capable of filtering out extreme isolated measurements, as demonstrated by Figure 2. In Figure 2(a) there are some measurements which have load classification $HIGH$, e.g. around $t = 1200$, which are filtered out in Figure 2(b). In addition, during the RUBiS case study, we have noticed known performance problems in RUBiS [16].

B. Automatability & Scalability

1) Automatability: We have manually executed all steps in our case studies. These steps are all easily automatable. An interesting problem is when to update the association rules. In the EOL case study we have shown that 3 months after training our rulesets were still able to identify performance anomalies, which leads to the expectation that the rules do not need regeneration often. An example of a situation in which it does need to be regenerated is after removing or adding a new server to the system. Our current solution is to retrain the rules with the new set of performance counters, but in future work we will research more sophisticated ways of doing this.

In our current case studies the length of the period during which training data was monitored was based on the availability of the data. More research should be done to define an ideal length for this period.

2) Scalability: In our case the EOL database was 17 GB in size. Preparing the data and training the association rules took approximately 1 hour. Classification of a new measurement took less than one second, which makes the approach scalable as the data preparation and training phase are executed rarely. For the RUBiS case study, the data preparation and training phase took two minutes. This means that the approach can easily be retrained on new data when necessary.

C. Threats to Validity

1) Different Applications: Both case studies were performed on SaaS applications, and we believe especially the

EOL case is representative of a large group of (multi-tenant) SaaS applications that can benefit from self-adaptiveness. While the RUBiS case is not representative for modern applications anymore [15], it is a widely-used benchmark in performance studies and a useful second validation of our approach.

Our approach is lightweight and transparent; it requires no modification of application code as measurements are done at the operating system level. In addition, our approach does not need knowledge about the structure of the system. A downside of our approach is that it does not give a diagnosis when performance problems arise. Until this is addressed in future work, our approach acts as a first indication that something is wrong in the system for more specialized algorithms which can *zoom in* to give better diagnosis [17].

Changing patterns in the functionality or workload of applications is a challenging problem [18]. More research should be done to make definite statements about how our approach responds to this.

2) Threats to Validity: We have already touched upon some of the issues concerning external validity in the above discussion. As far as the internal validity is concerned, we have performed 10-fold cross-validation on the EOL dataset to ensure the JRip algorithm used to generate the association rules generates stable rulesets on this type of data.

In our experimental setup we have used both industrial and synthetic workloads in our case studies. While we acknowledge that the synthetic workload may not provide a realistic load on the system, its main purpose was as a second validation after the industrial case study.

With respect to reliability, the SQL queries and Java code used for the training and load characterization phase are available for download from our website[8]. WEKA and JFreeChart are open source libraries for Java.

IX. Related Work

In load characterization research, many work focuses on detecting performance problems using low-level system measurements rather than giving a representation of the load on the system. In this section we give an overview of some of the most important work.

Cherkasova et al. [18] present an approach for deciding whether a change in performance was caused by a performance anomaly or a change in workload. Cohen et al. [7] present an approach to correlate low-level measurements with SLO violations. They use tree-augmented naive Bayesian networks as a basis for performance diagnosis and forecasting. In later work [19] they extend their approach to support multiple models to adapt to changing workloads. Both the approach used by Cohen et al. and Cherkasova et al. are similar to ours, but they use different statistics and they are only evaluated on synthetic data. The use of multiple models [19] can help improve our approach in the future.

[8]http://swerl.tudelft.nl/bin/view/Main/MTS

Part of existing research concerns the definition and adaptation of thresholds for low-level system measurements to detect performance problems. Breitgand et al. [20] were one of the first to propose an approach for automated threshold setting for components. In their approach, they set a threshold for the true positive and negative rate of the violation of a binary SLO. Based on this setting, their model tries to adapt the thresholds for components such that the true positive and negative rate converge to their threshold. In contrast to our work, the approach of Breitgand et al. uses single threshold values for performance metrics, while we use association rules which lead to combinations of thresholds for several metrics.

Magpie [21] is an approach which traces requests and their resource consumption in the system on event-level and uses that information to construct a workload model. Zhang et al. [22] propose the use of hardware-level performance counters (e.g. processor instruction count) to describe the workload of an application. The use of event or hardware-level information may form a valuable addition to our approach in the future, as we use operating system-level counters only currently.

Correa and Cerqueira [23] use statistical approaches to predict and diagnose performance problems in component-based distributed systems. For their technique, they compare decision tree, Bayesian network and support vector machine approaches for classifying. In contrast to our own work, Correa and Cerqueira's work focuses on distributed systems, making network traffic an important part of the equation.

X. Conclusion

In this paper we have investigated how to relate low-level performance measurements to high-level performance objectives. We have proposed an approach for characterizing the load on a system as either low, medium or high using performance counter measurements only, and we have evaluated this approach in two case studies. In short, our paper makes the following contributions:

- An approach for characterizing the load on a system using low-level performance measurements only
- An approach for detecting performance anomalies based on this load characterization
- Two case studies evaluating this approach
- An open source implementation of our approach as a toolbox in Java and SQL

Revisiting our research questions:

RQ 1 *Which metric characterizes system load and takes differences in performance requirements between applications and (groups of) users into account?* We have proposed the metric $SAratio$, which is the number of slow actions to normal actions per time interval.

RQ 2 *How can we relate (combinations of) low-level system measurements to this performance metric?* We

have shown that combinations of low-level system measurements can be related to a classification of the load on a system using statistics and association rule learning.

RQ 3 *How can performance anomalies be detected using these low-level system measurements?* In our evaluation we have shown that by calculating the $intensity$ of the load of a system using a sliding window approach, performance anomalies can be detected.

A. Future Work

In future work we will do more industrial case studies to evaluate our approach, including more research on more dynamic systems in which servers are added and removed. In addition, we aim to extend our approach with support for performance diagnosis, i.e., give an indication of which server or component causes a performance anomaly.

ACKNOWLEDGMENT

The authors would like to thank Exact for providing the funds and opportunity to perform this research. Further support came from the *NWO Jacquard ScaleItUp* project.

REFERENCES

[1] J. O. Kephart, "Research challenges of autonomic computing," in *Proc. Int'l Conf. on Softw. Engineering (ICSE)*. ACM, 2005, pp. 15–22.

[2] B. Simic, "The performance of web applications: Customers are won or lost in one second," http://www.aberdeen.com/aberdeen-library/5136/RA-performance-web-application.aspx, 2008, last visited: December 6th, 2011.

[3] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[4] C.-P. Bezemer and A. Zaidman, "Multi-tenant saas applications: maintenance dream or nightmare?" in *Proc. Joint Workshop on Softw. Evolution & Int. Workshop on Principles of Softw. Evolution (IWPSE/EVOL)*. ACM, 2010, pp. 88–92.

[5] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, 2009.

[6] R. Berrendorf and H. Ziegler, "PCL – the performance counter library: A common interface to access hardware performance counters on microprocessors," Central Institute for Applied Mathematics – Research Centre Juelich GmbH, Tech. Rep. FZJ-ZAM-IB-9816, 1998.

[7] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in *Proc. Symposium on Operating Systems Design & Impl.* USENIX Association, 2004, pp. 16–16.

[8] A. Webb, *Statistical Pattern Recognition*. Wiley, 2002.

[9] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann Publishers Inc., 2005.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.

[11] W. W. Cohen, "Fast effective rule induction," in *Proc. Int'l Conf. on Machine Learning*. Morgan Kaufmann, 1995, pp. 115–123.

[12] E. Fuchs and P. E. Jackson, "Estimates of distributions of random variables for certain computer communications traffic models," in *Proc. symposium on Problems in the optimization of data communications systems*. ACM, 1969, pp. 205–230.

[13] F. Dekking, C. Kraaikamp, H. Lopuhaa, and L. Meester, *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer, 2005.

[14] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of EJB applications," in *Proc. of the SIGPLAN conf. on OO-programming, systems, languages, and applications (OOPSLA)*. ACM, 2002, pp. 246–261.

[15] R. Hashemian, D. Krishnamurthy, and M. Arlitt, "Web workload generation challenges - an empirical investigation," *Software: Practice and Experience*, 2011, DOI: 10.1002/spe.1093.

[16] B. Pugh and J. Spacco, "Rubis revisited: why j2ee benchmarking is hard," in *Companion to the 19th annual SIGPLAN conf. on OO-programming systems, languages, and applications (OOPSLA)*. ACM, 2004, pp. 204–205.

[17] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf, "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *Int'l Conf. on Autonomic Computing (ICAC)*. ACM, 2011, pp. 141–150.

[18] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2008, pp. 452 –461.

[19] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2005, pp. 644–653.

[20] D. Breitgand, E. Henis, and O. Shehory, "Automated and adaptive threshold setting: Enabling technology for autonomy and self-management," in *Proc. Int'l Conf. on Autonomic Computing (ICAC)*. IEEE, 2005, pp. 204–215.

[21] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proc. Symposium on Operating Systems Design & Implementation*. USENIX Association, 2004, pp. 18–18.

[22] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen, "Processor hardware counter statistics as a first-class system resource," in *Proc. workshop on Hot topics in operating systems*. USENIX Association, 2007, pp. 14:1–14:6.

[23] S. Correa and R. Cerqueira, "Statistical approaches to predicting and diagnosing performance problems in component-based distributed systems: An experimental evaluation," in *Proc. of the Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2010, pp. 21–30.

TUD-SERG-2012-006
ISSN 1872-5392