Performance Optimization of Deployed Software-as-a-Service Applications

Cor-Paul Bezemer[1], Andy Zaidman[1]

[a]*Delft University of Technology, Faculty EEMCS, Mekelweg 4, 2628 CD Delft, The Netherlands*

Abstract

The goal of performance maintenance is to improve the performance of a software system after delivery. As the performance of a system is often characterized by unexpected combinations of metric values, manual analysis of performance is hard in complex systems. In this paper, we propose an approach that helps performance experts locate and analyze spots – so called performance improvement opportunities (PIOs) –, for possible performance improvements. PIOs give performance experts a starting point for performance improvements, e.g., by pinpointing the bottleneck component. The technique uses a combination of association rules and performance counters to generate the rule coverage matrix, a matrix which assists with the bottleneck detection.

In this paper, we evaluate our technique in two cases studies. In the first, we show that our technique is accurate in detecting the timeframe during which a PIO occurs. In the second, we show that the starting point given by our approach is indeed useful and assists a performance expert in diagnosing the bottleneck component in a system with high precision.

Keywords: performance maintenance, performance analysis

1. Introduction

In the ISO standard for software maintenance[1], four categories of maintenance are defined: corrective, adaptive, perfective and preventive maintenance. Perfective maintenance is done with the goal of improving and therefore perfecting a software system after delivery. An interesting application of perfective maintenance is *performance maintenance*, which is done to enhance the performance of running software by investigating and optimizing the performance after deployment [**?**]. A reason to do this after deployment is that it may be too expensive to create a performance testing environment that is equal to the production environment, especially for large systems. As an example, many Software-as-a-Service (SaaS) providers spend a fair portion of their budget each month on hosting infrastructure as infrastructure forms the most important factor in the total data center cost [**?**]. Copying the production system to provide an environment for performance testing will further increase these costs. Therefore, it is sometimes necessary to analyze and adapt the deployed system directly.

While a large amount of research has been done on software performance engineering in general [**?**], only few papers deal with software performance maintenance. Performance maintenance poses different challenges, as we are dealing with live environments in which computing resources may be limited when we are performing maintenance. In addition, experience from industry shows that performance maintenance engineers mainly use combinations of simple and rather inadequate tools and techniques rather than integrated approaches [**?**], making performance maintenance a tedious task.

Perfecting software performance is typically done by investigating the values of two types of metrics [**?**]. On one hand, high-level metrics such as response time and throughput [**?**] are important for getting a general idea of the performance state of a system. On the other hand, information retrieved from

[1] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39064

lower-level metrics, e.g., metrics for memory and processor usage — so called performance counters [**?**] —, is important for pinpointing the right place to perform a performance improvement. However, determining a starting point for analysis of these lower-level metrics is difficult, as the performance of a system is often characterized by unexpected combinations of performance counter values, rather than following simple rules of thumb [**?**]. This makes manual analysis of performance in large, complex and possibly distributed systems hard.

In this paper, we present a technique which provides assistance during semi-automated performance analysis. This technique automates locating so-called *performance improvement opportunities (PIOs)*, which form a starting point for analysis of performance counters. Interpreting the results of automated performance analysis approaches is difficult for human experts [**?**]. Our approach aims to assist experts by analyzing these starting points to give a diagnosis of bottleneck component(s). In short, we focus on the following research questions:

- **RQ 1** *How can performance counter values provide assistance during the performance optimization process?*
- **RQ 2** *How can we interpret these values so that they can lead to the identification of the bottleneck component(s) of a system?*

In previous work, we have done a preliminary evaluation of this technique by performing a user study on an industrial SaaS application [**?**]. During this preliminary evaluation, we demonstrated the feasibility of our approach and its applicability in industry for assisting during semi-automated performance analysis. In this work, we first show that our technique is accurate in detecting the timeframe during which a PIO occurs. In a second case study, we show that the starting point given by our approach is indeed useful and assists a performance expert in diagnosing the bottleneck component in a system with high precision.

This paper is organized as follows. In Section **??**, we introduce the concept of PIOs and we present our approach for detecting such PIOs. In Section **??**,

we explain our approach for automatically analyzing these PIOs. Section **??** discusses the implementation of our approach. Our case studies are presented in Sections **??**, **??** and **??**. We discuss the results of these case studies and threats to the validity of these results in Sections **??** and **??**. We present related work in Section **??** and we conclude our work in Section **??**.

2. Detecting Performance Improvement Opportunities

Performance optimization can be done during the software design phase and after deployment. Techniques such as profiling [**?**] can be used by the developer to find and fix application bottlenecks during the design phase. However, these techniques cannot always be used after deployment, as they are usually very expensive and not capable of dealing with complex systems which are deployed on multiple servers [**?**]. Therefore, it is necessary to use more light-weight techniques after deployment to optimize system performance.

In order to start our investigation on how we can improve the performance of a system that is deployed, we must be able to do the following:

- *Requirement 1* Detect the timeframes during which the system performed relatively slow, i.e., find situations in which performance optimization is possible.
- *Requirement 2* Detect the component(s) that is/are the bottleneck component(s).

By knowing at least this, we have a starting point for our investigation of optimizing the performance of a deployed system. In the remainder of this paper, we present our approach for detecting these requirements automatically from performance data. In the next section, we introduce so-called *performance improvement opportunities* to assist performance experts in their investigation on performance optimization. In Section **??**, we will present our approach for detecting these PIOs (*Requirement 1*). We will explain our approach for analyzing PIOs (*Requirement 2*) in Section **??**.

2.1. Performance Improvement Opportunities (PIOs)

A *performance improvement opportunity* (PIO) is a collection of performance data collected during a period of time at which the performance of the system could possibly be improved. It is a description of a situation in a system in which performance optimization may be possible. A PIO contains info needed to analyze the situation during which it was detected:

- Date and time of start of the PIO
- `SARatio` metric (Section **??**)
- Intensity transformation (Section **??**)
- Rule coverage matrix (Section **??**)

A PIO description can assist engineers in performing perfective maintenance by pinpointing the bottleneck component during the PIO. The next step could be investigation of that component using a profiler (see Section **??**). When we improve the performance of a system using the information in a PIO, we say we *exploit* the PIO. Throughout this paper we will use the term PIO and PIO description interchangeably.

2.2. `SARatio` Metric

Application performance can be expressed in many different metrics, such as response time, throughput and latency [**?**]. One of the most important is average response time [**?**], 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[2] and/or users. Therefore, it may exclude details about the performance state of a system, details that can be important for detecting a performance improvement opportunity.

[2] An action is the activation of a feature by the user. A feature is a product function as described in a user manual or requirement specification [**?**].

An example of this is 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.

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 generation) 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.

To define a metric which fits into this description, we propose to refine the classical response time metric so that we take into account the difference between actions and users. In order to do so, we 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, the metric that we are constructing becomes 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 has approximately the same response time when executed by the same user at different times (see Table **??**).

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

$$\texttt{SARatio}_t = \frac{|SLOW_t|}{|SLOW_t| + |NORMAL_t|}$$

Because it is a ratio, isolated extreme values have a smaller influence on the metric, which makes it more independent of time interval[3].

We distinguish three groups of values for `SARatio`:

- *HIGH* - the 5% highest values, indicating the times at which the system is relatively the slowest and therefore the most interesting for performance optimization
- *MED* - the 10% medium values
- *LOW* - the 85% lowest values

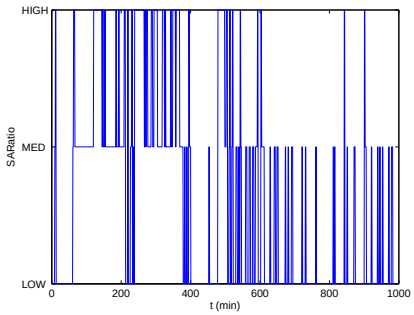
As a threshold for the *MED* and *HIGH* classes we use the 85^{th} and 95^{th} percentile of the distribution of `SARatio`. By using the 85^{th} and 95^{th} percentile we use approximately the same confidence intervals as commonly used for the normal distribution [**?**]. For an industrial system, i.e., 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.

Throughout the rest of this paper, we will refer to the $HIGH$, MED and LOW values for `SARatio` as $classifications$. All time periods containing *HIGH* values for `SARatio` constitute possible PIOs and therefore require deeper investigation. In order to focus on the greatest performance improvements possible, we would like to investigate longer lasting PIOs first. Figure **??** shows an example graph of 1000 minutes of `SARatio` values. This graph has several disadvantages:

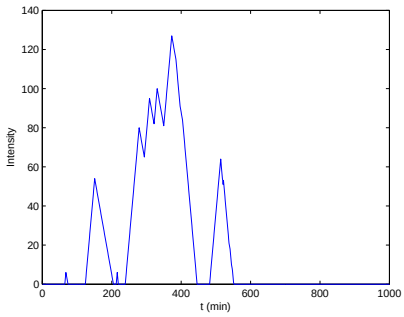
- It becomes unclear when large (e.g. $t > 1000$) periods of time are displayed
- It is difficult to distinguish longer lasting PIOs from shorter lasting ones

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

We transform Figure **??** into Figure **??** by using the intensity transformation discussed in the next section. The goal of this transformation is to show a clear graph in which it is easy to detect longer lasting PIOs.



(a) Before



(b) After

Figure 1: `SARatio` graph before and after intensity transformation

2.3. Intensity Transformation

Intuitively, we expect that we can achieve greater performance improvements by investigating longer lasting PIOs. The rationale behind this intuition can be explained by the following example. In a system, a PIO of 60 minutes and a PIO of 10 minutes are detected. As it is likely that more customers will be affected by the relatively slow performance of the system during the PIO of 60 minutes, we would like to investigate this PIO first.

Therefore, we would like to emphasize the occurence of high `SARatio` values which are close to each other in time, i.e., longer lasting PIOs. To make

such occurences easier to spot, we perform the transformation described in Algorithm **??** on the `SARatio` data. This transformation uses a sliding window approach to emphasize longer lasting PIOs.

A window of size n contains the `SARatio` classifications of the last n time frames. We count the occurrences of LOW, MED and $HIGH$ classifications and keep a counter *intensity*. Every time the majority ($\geq 33\%$) of the classifications in the window are $HIGH$, i.e., the system is relatively slow, *intensity* is increased by 2. When the system returns to normal performance, i.e., the majority of the classifications in the window are MED or LOW, *intensity* is decreased by 1 and 2 respectively. These steps are depicted by Algorithm **??** (INTENSITYTRANSFORMATION). Figure **??** shows the effect of applying this transformation to the data in Figure **??**. It is easy to see that there are three major PIOs in Figure **??**. Note that it is easy to automate the process of locating PIOs by setting the start of a PIO whenever the *intensity* becomes larger than a certain threshold. Throughout this paper, we assume the *intensity* threshold is 0.

3. Analyzing Performance Improvement Opportunities

Now that we have a technique for detecting PIOs, the next step is to analyze them. In our approach for PIO analysis we use the `SARatio` described in the previous section as a foundation for training a set of association rules [**?**] which

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

Require: Window size n, a set of `SARatio` 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: $\quad intensity = intensity + 2$
8: **else if** $maxCnt == cntMed$ **then**
9: $\quad intensity = max(intensity - 1, 0)$
10: **else**
11: $\quad intensity = max(intensity - 2, 0)$
12: **end if**
13: **return** *intensity*

help us analyze the PIO. We use association rules because they make relationships in data explicit, allowing us to use these relationships in our analysis.

In this section, we will explain how these association rules can assist us in analyzing PIOs and how we generate them.

3.1. PIO Analysis Using Assocation Rules

The goal of analyzing PIOs is to find out which component forms the bottleneck. This component can then be replaced or adapted to optimize the performance of the system. *Performance counters* [**?**] (or *performance metrics*) offer easy-to-retrieve performance information about a system. These performance counters exhibit details about the state of components such as memory, CPU and web servers queues and therefore we would like to exploit this information to decide which component(s) form the bottleneck. An important observation is that the performance of a system often is characterized by unexpected combinations of performance counter values, rather than following simple rules of thumb [**?**]. Therefore, we cannot simply detect a bottleneck component using a threshold for one performance counter. It is our expectation that throughout a PIO, we can detect clusters of performance counter values which point us in the direction of the bottleneck component(s).

Performance analysis is usually done on high-dimensional data, i.e., many performance counters, and analysis of this data is not trivial. In addition, understanding the results of automated analysis is often difficult. Therefore, we propose to use visualization as a foundation for our PIO analysis approach. The requirements of such a visualization technique are:

- It must allow easy detection of clusters of performance counters
- It must be capable of displaying high-dimensional data

A visualization technique which fulfills these requirements is the heat map [**?**]. Figure **??** depicts an example of a heat map, which could assist during performance analysis. The heat map displays data for two performance counters (*PC1* and *PC2*) monitored on five servers (*S1* – *S5*). In this heat map, darker

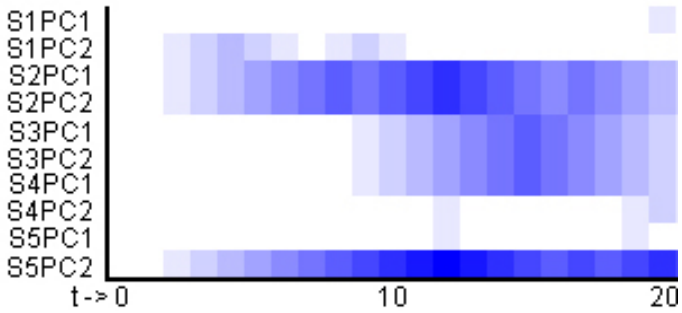


Figure 2: Rule Coverage Heat Map

squares mean that there is a stronger indication that the component on which this performance counter was monitored forms a bottleneck. In Figure **??** it is easy to see that server *S2* and performance counter *PC2* on server *S5* require deeper investigation. In addition, a heat map is capable of displaying high-dimensional data because every performance counter is represented by one row in the heat map. As the rows do not overlap, the visualization is still clear for high-dimensional data.

3.1.1. The Rule Coverage Matrix

The heat map in Figure **??** is a direct visualization of the *rule coverage matrix* depicted by Table **??**. The rule coverage matrix contains information which helps us detect clusters of performance counters causing a PIO. In the remainder of this paragraph we will explain how association rules help us to generate this matrix.

Table 1: Rule Coverage Matrix for Figure **??**

	t =**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
S1PC1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
S1PC2	0	0	1	2	3	2	1	0	1	2	1	0	0	0	0	0	0	0	0	0	0
S2PC1	0	0	1	2	3	4	5	6	7	6	7	8	9	8	7	6	5	6	5	4	3
S2PC2	0	0	1	2	3	4	5	6	7	6	7	8	9	8	7	6	5	6	5	4	3
S3PC1	0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	6	5	4	3	3
S3PC2	0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	6	5	4	3	2
S4PC1	0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	6	5	4	3	2
S4PC2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	2
S5PC1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
S5PC2	0	0	1	2	3	4	5	6	7	8	9	10	11	10	9	8	7	8	7	8	9

3.1.2. Association Ruleset Generation

During the *association ruleset generation* (or *training*) phase, we collect and analyze logged actions and performance data of a system and build a set of association rules. An example association rule could be:

$$\texttt{CPU\_Utilization} \geq 80\% \ \texttt{\&} \ \texttt{Memory\_Usage} \geq 50\% \rightarrow \texttt{HIGH}$$

This rule signals that during the training phase we observed that if the CPU is used at 80% or more and the memory is used for at least 50% there was a significant slowdown in the system, i.e., the `SARatio` was $HIGH$.

To generate such association rules, we monitor the performance counters and log all actions during a training period. The set of interesting performance counters is different for different systems and different applications. Therefore, we advise to monitor a large set of performance counters initially, and to narrow down the set to monitor after generating the association rules later. After this, we calculate the `SARatio` for every time frame in the action log and use this together with the monitored performance counter data as input for the association rule generation algorithm. The result of this will be association rules that will take performance counter values as input and output `SARatio` classifications. In this way, we bridge the low level performance counters to a `SARatio` classification. This allows us to monitor the performance counters and then use them for a) PIO location and b) PIO analysis.

3.1.3. Rule Coverage Matrix Generation

Our approach for rule coverage matrix generation uses a matrix m with one row for each performance counter and one column for every time t we receive a measurement. This matrix contains the raw values monitored for each counter. Because performance analysis is difficult to do on raw performance counter values, we maintain a so-called rule coverage matrix m_{rcm} to assist during performance analysis. The rows of this matrix contain the performance counters, the columns depict measurements of performance counters. Every measurement contains all performance counter values monitored in a certain time interval. The first column, representing the first measurement is initialized

to 0. Each time a new measurement is received, the last column of m_{rcm} is copied and the following algorithm is applied:

- Increase $m_{rcm}^{i,j}$ by 1 if performance counter i is covered by a *high* rule at measurement j.
- Leave $m_{rcm}^{i,j}$ equal to $m_{i,j-1}$ for a *med* rule
- Decrease $m_{rcm}^{i,j}$ by 1 if performance counter i is covered by a *low* rule at measurement j, with a minimum of 0

Note that the original 'raw' values of the performance counters in m are left untouched in this process. We update the value of every $m_{rcm}^{i,j}$ only once for every measurement, even though multiple covering rules may contain the same performance counter.

The rationale behind building the rule coverage matrix this way is the following:

1. The ruleset describes all known cases of when the system was performing slowly.
2. We expect all measurements made during a PIO to be covered by the same, or similar rules when they are classified. The reason for this is that we expect that abnormal values of (combinations of) performance counters will be exhibited for a longer period of time, i.e., throughout the PIO.
3. When entering this into the rule coverage matrix as described, higher values in m_{rcm} will appear because these values will be increased for performance counters which occur in adjacent measurements.
4. Eventually, clusters of higher values in m_{rcm} for performance counters for specific components will appear.
5. These clusters can be used to do performance maintenance, e.g., by pinpointing a bottleneck component.

The following example illustrates this. Figure **??** shows the resulting m_{rcm} after applying our approach to the measurements and ruleset of Table **??** for a system consisting of two servers *S1* and *S2*, which are monitored through three performance counters (*S1PC1*, *S1PC2* and *S2PC1*). The first column depicts the situation after the measurement done at $t = 0$. This measurement fires rule 0, which does not include any performance counters, leaving all values in the rule coverage matrix untouched. The measurement made at $t = 1$ fires rule 3, hence increasing only the value for *S1PC1*. Continuing this procss results in the matrix depicted by Figure **??**.

Figure **??** shows the heat map of this matrix. In our simple example we can see a cluster of dark coloured performance counters at server *S1*, indicating this server may be a bottleneck.

Table 2: Sample ruleset and performance measurements

Sample association ruleset				**Sample measurements**
1 S1PC1>80 & S2PC1<60 → *high*	t	S1PC1	S1PC2	S2PC1
2 S1PC1>70 & S1PC2>70 → *high*	0	40	60	80
3 S1PC1>90 → *high*	1	95	60	80
4 S1PC2<30 → *med*	2	98	80	80
5 else → *low*	3	98	95	55
	4	98	80	80
	5	40	45	80

	0	1	2	3	4	5
S1PC1	0	1	2	3	4	4
S1PC2	0	0	1	2	3	3
S2PC1	0	0	0	1	0	0
covered by rules #	5	3	2,3	1,2,3	2,3	5

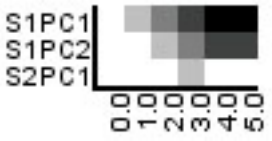


Figure 3: Rule coverage matrix for Table **??** and the corresponding heatmap

As association rule learning is a form of supervised learning, it is possible that the generated association ruleset does not cover all PIOs. This is inherent to the characteristics of supervised learning, as such learning algorithms generate classifiers which are specialized at detecting cases that have occurred during the training phase. In future work, we will investigate how to improve the quality of the generated association rule set.

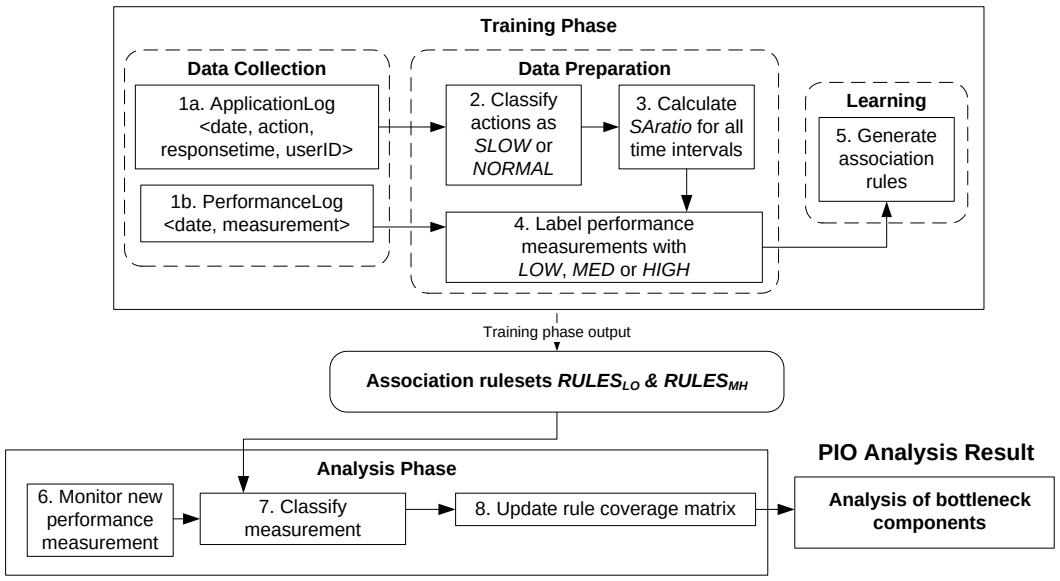


Figure 4: Steps of our approach for analyzing PIOs

In the next section we will discuss the implementation of our approach.

4. Implementation

Figure **??** depicts all the steps required for the implementation of our approach. In this section, we will explain every step taken.

4.1. Training Phase

During the training phase (see Section **??**) the association rules used for PIO analysis are generated. First, we collect the required data and calculate the `SARatio` for every time frame. Then, we generate the association ruleset.

4.1.1. Data Collection

We log all actions in the system, including their response time and the ID of the user that made them, for a period of time (*step 1a* in Figure **??**). 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 (1) date, (2) action, (3) responseTime and (4) userID (if existent) for every action made to the application
- A log file `PerformanceLog` containing (1) low-level system performance measurements and the (2) date at which they were made

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`).

4.1.2. Data Preparation

After collecting the data, we classify all actions in the `ApplicationLog` as *slow* or *normal* (*step 2*) and calculate the $\texttt{SARatio}_t$ per time interval t as described in Section **??** (*step 3*). We label all low-level measurements in the `PerformanceLog` with their corresponding load classification (*step 4*).

4.1.3. Learning

The final step of the training phase is to apply the association rule learning algorithm to the labeled data (*step 5*). Because the LOW class is much larger than the MED and $HIGH$ classes, we generate a random subset of the LOW class, which is approximately equal in size to the number of MED plus the number of $HIGH$ elements. This helps us to deal with the problem of overfitting [**?**], and improves the classification result as the result will not be biased towards the LOW class anymore.

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.

We combine the MED and $HIGH$ classes into the temporary $OTHER$ class and use the random subset of the LOW 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 $RULES_{MH}$.

4.2. Analysis Phase

During the analysis phase, 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*). After the classification is done, the rule coverage matrix is updated (*step 8*). Finally, this matrix can be used to analyze performance improvement opportunities.

5. Experimental Setup

The goal of our evaluation is to show that our approach is capable of fulfilling the two requirements posed in Section **??**, namely detecting the timeframes during which the system performed relatively slow and detecting the bottleneck components. To do this evaluation, we propose two case studies. In case study I (Section **??**), we will show that the `SARatio` is an accurate metric for detecting timeframes during which the system was slow. In addition, in this case study we will show that our technique is capable of estimating the `SARatio` classifications using performance counter measurements. In case study II (Section **??**), we will use the knowledge of a performance expert to manually verify the classification results of our approach. This verification will show that our approach is capable of detecting bottleneck components.

Hence, in these case studies we address the following research questions:

1. Is the `SARatio` an accurate metric for detecting the timeframes during which the system was slow? *(Case study I)*
2. Is our technique for the estimation of `SARatio` classifications using performance counter measurements accurate? *(Case study I)*

3. How well do our the results of our PIO analysis approach correspond with the opinion of an expert? *(Case study II)*

In this section, the experimental setup of the case studies is presented.

5.1. Case Study Systems

We performed two case studies on SaaS systems: (1) on a widely-used benchmark application running on one server (RUBiS [**?**]) and (2) on a real industrial SaaS application running on multiple servers (Exact Online [**?**]).

5.1.1. RUBiS

RUBiS is an open source performance benchmark 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.

5.1.2. Exact Online

Exact Online is an industrial multi-tenant SaaS application for online bookkeeping with approximately 18,000 users[4]. Exact Online is developed by Exact, a Dutch-based software company specializing in enterprise resource planning (ERP), customer relationship management (CRM) and financial administration software. The application currently runs on several web, application and database servers. It is written in VB.NET and uses Microsoft SQL Server 2008.

5.2. Process

Training Phase. The `ApplicationLog` and `PerformanceLog` are collected using the webserver and OS-specific tools and are imported into a SQL database; all steps in the data preparation phase are performed using a sequence of SQL queries. The generation of the *LOW*, *MED*, *HIGH* classes is done by custom

[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.

implementation in Java. For the implementation of the rule learning algorithm we have used the JRip class of the WEKA API [**?**], which is an implementation of the RIPPERk algorithm [**?**]. 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 datasets with respect to classification error and speed.

Analysis Phase. The steps performed during the analysis phase are implemented in Java, resulting in a tool that can be used on newly monitored data. The rule coverage matrix is generated with the help of the WEKA API. The visualizations used for PIO analysis are generated using JFreeChart [**?**] and JHeatChart [**?**].

6. Proof-of-Concept: Case Study I for `SARatio` Classification Estimation Using Performance Counter Measurements

Our PIO analysis approach relies on the rule coverage matrix. To build this matrix, we use a combination of association rules and performance counters to estimate `SARatio` classifications. As a proof-of-concept, we have verified that this combination is indeed a solid foundation for estimating the `SARatio` classification in the following settings:

- On a simulated PIO in RUBiS
- On a real PIO in EOL

In this section, these proof-of-concept studies will be presented.

6.1. RUBiS

The goals of the RUBiS proof-of-concept were as follows:

- To show that our approach can closely estimate the `SARatio` caused by synthetically generated traffic
- To show that it is capable of dealing with problems on the client side, i.e., that it does not recognize client side problems as PIOs

In order to generate several traffic bursts, we have configured 3 RUBiS workload clients to run for 30 minutes in total. Figure **??** 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. This level was reached around $t = 800$, causing a slowdown on the clients which resulted in less traffic generated. Due to the implementation of RUBiS which uses synchronous connections [**?**], i.e., the client waits for a response from the server after sending a request, the response times went up. Because 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, this overload situation also resulted in higher durations in the `ApplicationLog`. However, this increase in response times is not caused by a server problem (i.e., noticeable from performance counter values), hence we expect our approach to convert performance counter measurements at that time to low `SARatio` values.

6.1.1. Training Phase

Data Collection. Table **??** shows the set of performance counters monitored on the server; we used Dstat [**?**] to log them every second. Together with the Apache *access_log*, we could now create the SQL databases `ApplicationLog` and `PerformanceLog`. These databases have the same structure as those in the EOL proof-of-concept so that the same queries could be used. Table **??** contains some statistics about the collected data.

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. Table **??** shows the number of *slow* and *normal* requests for these applications. Figure **??** shows the distribution of `SARatio` for the RUBiS case study, together with the 85^{th} and 95^{th} percentile.

Learning. Performing the association rule learning algorithm resulted in a ruleset $RULES_{LO}$ of 6 rules and a ruleset $RULES_{MH}$ of 2 rules. Table **??** shows the generated rules.

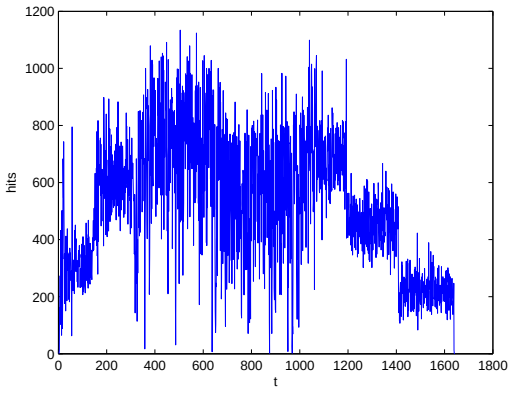


Figure 5: Traffic generated for the RUBiS case study

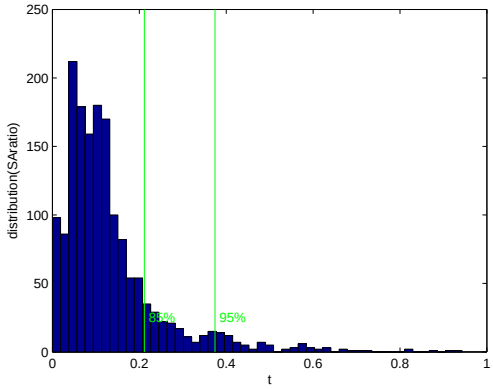


Figure 6: Distribution of `SARatio` for 30 minutes of RUBiS traffic

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 3: Monitored performance counters for RUBiS

Table 4: Association rules generated in the RUBiS proof-of-concept

$RULES_{LO}$
(mem/cach $\leq$ 2175963136) & (mem/used $\geq$ 1103503360) $\rightarrow$ OTHER (mem/cach $\geq$ 1910624256) & (mem/buff $\leq$ 316026880) $\rightarrow$ OTHER (mem/buff $\geq$ 316256256) & (mem/buff $\leq$ 316358656) & (system/int $\leq$ 6695) & (dsk/read $\geq$ 118784) $\rightarrow$ OTHER (mem/buff $\leq$ 316497920) & (system/int $\geq$ 7052) & (mem/used $\leq$ 1080979456) $\rightarrow$ OTHER (mem/cach $\leq$ 2215194624) & (dsk/read $\leq$ 24576) $\rightarrow$ OTHER else $\rightarrow$ LOW

$RULES_{MH}$
(filesystem/files $\leq$ 2336) $\rightarrow$ HIGH else $\rightarrow$ MED

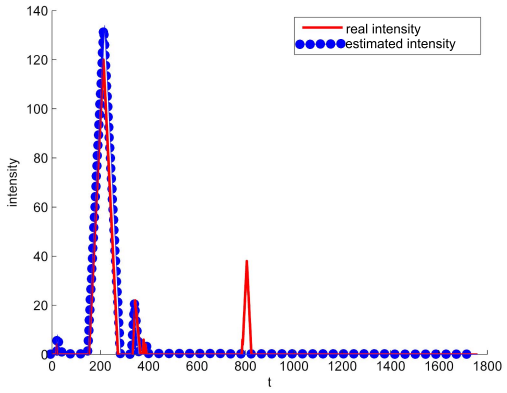


Figure 7: Real intensity versus estimated intensity

6.1.2. Analysis Phase

To validate our approach for the RUBiS case study we 1) calculated the intensity directly from the `ApplicationLog` using the `SARatio` and 2) estimated the intensity using assocation rules. The rationale behind step 2 is that we need to estimate the `SARatio` classifications using association rules before we can estimate the intensity. If the estimated intensity then matches with the intensity calculated during step 1, we have a validation that our approach for estimating the `SARatio` using performance counter measurements yields correct results for the RUBiS case. Because the association rules were generated from a subset of the `PerformanceLog` as described in Section **??**, 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, approximately 67% of the data analyzed during the analysis phase was new. Figure **??** shows the graph of the real and estimated intensity[5].

6.1.3. Evaluation

The graphs for the real and estimated intensity are nearly equal, except for one peak. As expected, the real intensity shows a peak around $t = 800$ due to increased response times, caused by the synchronous connections, whereas the estimated intensity does not. An interesting result of this is that while the real intensity will falsely detect a PIO, the estimated intensity ignores this, which is correct. The peak around $t = 200$ can be explained by the fact that the workload client executes certain heavy search queries for the first time. After this the results are cached, resulting in less load on the server. The intensity estimation was capable of detecting this.

Revisiting the goals stated in the beginning of this section, the RUBiS proof-of-concept shows our approach is capable of estimating the `SARatio` classifications well, as demonstrated by Figure **??**. In fact, our approach is more precise than the approach that relies on the average response time directly, as our ap-

[5]This graph is best viewed in colour.

proach did not classify the overloaded client as a server slowdown.

Another interesting observation is that our approach was capable of detecting several known weaknesses in the RUBiS implementation [**?**], namely the fact that it uses synchronous connections for the communication between the client and the server, and the slow caching of the search queries at the beginning of the benchmark.

6.2. Exact Online

The goals of the Exact Online proof-of-concept were as follows:

- To show that our approach can closely estimate the `SARatio` caused by real traffic
- To show that our approach can detect PIOs in a period different than the period used to train the association rules
- To show that we can estimate the `SARatio` during unexpected events

We have analyzed 64 days of data which was monitored during the execution of Exact Online. During this period, a performance incident was caused by a bug in a product update. This bug caused logfiles to be locked longer than necessary, which resulted in bad performance.

As a proof-of-concept, we:

- Generated the association rulesets using data which was recorded 3 months before the incident, to show that we do not need to constantly retrain our rulesets
- Estimated the `SARatio` classifications during this incident using performance counters, to show that our approach is capable of estimating the `SARatio` during unexpected events

6.2.1. Training Phase

Data Collection. Exact Online performance data is stored for a period of 64 days in the form of logged performance counter values. Table **??** depicts the

subset of performance counters which are being logged. This list was selected by Exact performance experts, who had at least 7 years of experience with performance maintenance, and contains the 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 **??** shows some details about the collected data.

The `ApplicationLog` was retrieved by selecting the required elements from the Internet Information Server log. The performance measurements were logged into a database called `PerformanceLog` by a service which collects performance

	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 5: Details about the case studies

Virtual Domain Controller 1 & 2, Staging Server	
Processor\%Processor Time (*60s*)	
Service 1 & 2	
Memory\Available Mbytes (*300s*)	Process\%Processor Time (*30s*)
Processor\%Processor Time (*60s*)	System\Processor Queue Length (*60s*)
SQL Cluster	
LogicalDisk\Avg. Disk Bytes/Read (*30s*)	LogicalDisk\Avg. Disk Read Queue Length (*30s*)
LogicalDisk\Avg. Disk sec/Read (*30s*)	LogicalDisk\Avg. Disk sec/Write (*30s*)
LogicalDisk\Avg. Disk Write Queue Length (*30s*)	LogicalDisk\Disk Reads/sec (*30s*)
LogicalDisk\Disk Writes/sec (*30s*)	LogicalDisk\Split IO/sec (*60s*)
Memory\Available Mbytes (*60s*)	Memory\Committed Bytes (*300s*)
Memory\Page Reads/sec (*30s*)	Memory\Pages\sec (*30s*)
Paging File\%Usage (*60s*)	Processor\%Processor Time (*30s*)
Buffer Manager\Lazy writes/sec (*60s*)	Buffer Manager\Buffer cache hit ratio (*120s*)
Buffer Manager\Page life expectancy (*60s*)	Databases\Transactions/sec (*60s*)
Latches\Average latch wait time (ms) (*30s*)	Latches\Latch Waits/sec (*30s*)
Locks\Lock Waits/sec (*120s*)	Memory Manager\Memory grants pending (*60s*)
General Statistics\User Connections (*60s*)	SQL Statistics\Batch requests/sec (*120s*)
SQL Statistics\SQL compilations/sec (*120s*)	virtual\vfs_avg_read_ms (*60s*)
Webserver 1 & 2	
ASP.NET\Requests Current (*60s*)	ASP.NET\Requests Queued (*60s*)
ASP.NET Apps\Req. Bytes In Total (*120s*)	ASP.NET Apps\Req. Bytes Out Total (*120s*)
ASP.NET Apps\Req. in App Queue (*60s*)	ASP.NET Apps\Requests Total (*60s*)
ASP.NET Apps\Req./sec (*120s*)	Memory\Available Mbytes (*120s*)
Process\%Processor Time (*30s*)	Process\Handle Count (*60s*)
Process\Thread Count (*60s*)	Processor\%Processor Time (*60s*)

Table 6: Monitored performance counters for EOL (*measurement interval*)

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, and were in the range from 30 seconds to 10 minutes, depending on the counter. The configured interval for every counter is depicted by Table **??**.

Data Preparation. 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 [**?**]. Table **??** 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. 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 [**?**].

Figure **??** shows the distribution of `SARatio` in the EOL case study, together with the 85^{th} and 95^{th} percentile.

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 7: #actions per class for the selected samples

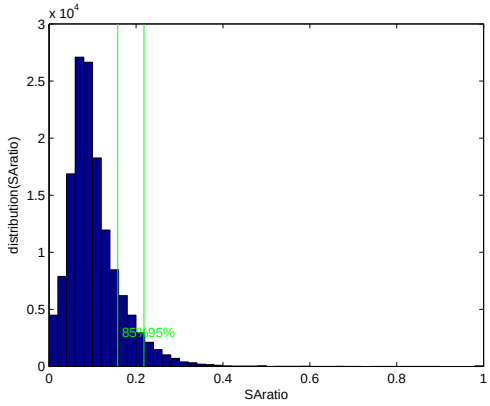


Figure 8: Distribution of `SARatio` for 64 days of EOL traffic

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[6].

6.2.2. Analysis Phase

We analyzed an incident that happened 3 months after the training data was recorded, which makes it a strong proof-of-concept as the training data and incident data are not biased towards each other. To validate the rule-sets, we have estimated the `SARatio` classifications using performance counter measurements. Figure **??** 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 $t = 4900$.

6.2.3. Evaluation

Figure **??** shows a high peak from approximately $t = 4100$ to $t = 4900$, which indicates our approach is capable of estimating the `SARatio` during unexpected events. 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

[6]Due to space limitations, we did not include the rules in this paper, but the complete set of rules can be viewed at http://www.st.ewi.tudelft.nl/~corpaul/data/assocrules_eol.txt.

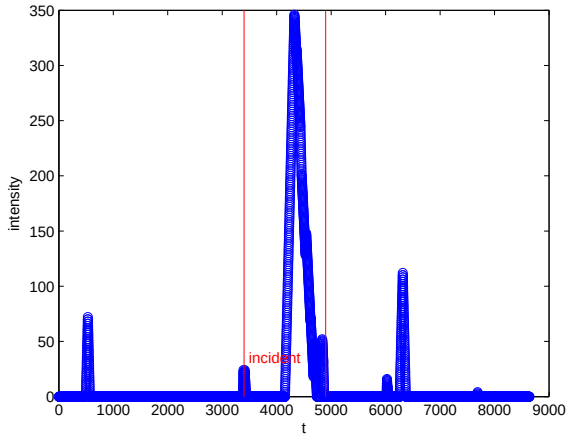


Figure 9: Intensity graph of the EOL incident based on estimated `SARatio` classifications

peaks were caused by heavier system load during administrative tasks such as database maintenance, which are performed at night for EOL.

As a comparison, Figure **??** shows the performance anomaly criterium used by the EOL team. In this criterium, an anomaly is reported when the average response time in an hour exceeds 450ms. Figure **??** 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

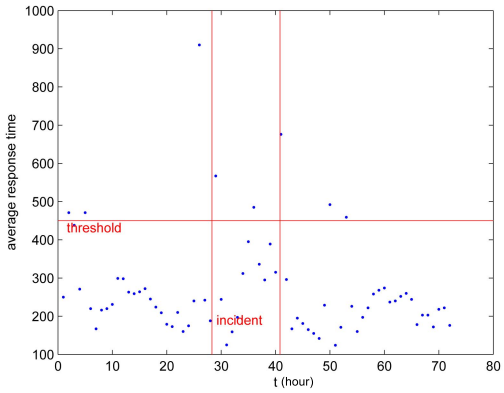


Figure 10: Current EOL performance anomaly criterium during incident

reported the anomaly between $t = 34$ and $t = 35$.

Revisiting the goals presented earlier in this section, the EOL case study shows that the `SARatio` can be estimated 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.

7. Case Study II: Evaluation of Exact Online Analysis Results

In this section, we present our case study in which we did an evaluation of PIO analysis results of our approach on industrial data. We address the following research question:

- How well do the results of our PIO analysis approach correspond with the opinion of an expert?

7.1. Case Study Description

Evaluating the precision of our PIO analysis approach is not trivial. Due to the nature of our approach, which is to assist experts in their manual analysis, the analysis results must be evaluated manually as well.

We have analyzed 66 days of data monitored during normal execution of Exact Online. During this period, 271 performance counters were monitored every minute on a total of 18 servers. These performance counters were collected

and stored in a centralized performance logging database. Note that the dataset was different from the one used in Section **??**. Because the server setup for EOL changed and became more complex since the case study described in that section, we decided to analyze the data for the new setup as we expected this to yield more interesting results.

Over the period of 66 days, 236 PIOs were located in total using our approach. Because manual analysis with the help of an expert is time-consuming and expensive, we verified only a random sample of this total set. In addition, a false negative analysis (i.e., missed PIOs) is difficult as we do not have a complete list of true PIOs for real data. Therefore, we extended our list of detected PIOs with overload registrations made using the overload detection rule currently used by engineers of EOL (see Section **??**). This rule will register any hour during which the average response time was larger than 450ms as a system overload (which is a form of a PIO). We manually verified a random sample of these overload registrations as well, with the goal of getting an indication of the number of false negatives of our approach. Table **??** depicts the number of detected PIOs and overload registrations and the size of the random sample.

Our approach	
Total # PIOs	236
# random sample	12 (5.1%)
Average duration per hour	
Total # registrations	182
# random sample	5 (2.8%)

Table 8: Random sample description

7.2. Training Phase

The training phase for this case study was equal to the process described in Section **??**, with a different data set for the training period. For this case study, the training data was monitored one week before the analyzed data. The result of the training phase are rulesets $RULES_{LO}$ and $RULES_{MH}$ depicted by Table **??**.

$RULES_{LO}$
(DBclus1/Processor/% Processor Time/_Total $\geq$ 13.02883) & (DBclus1/SQLServer:Latches/Latch Waits/sec/null $\geq$ 598.898376) $\rightarrow$ OTHER (DBclus1/Processor/% Processor Time/_Total $\geq$ 13.378229) & (DBclus1/SQLServer:Buffer Manager/Page life expectancy/null $\leq$ 1859) $\rightarrow$ OTHER (ws2/Processor/% Processor Time/_Total $\geq$ 11.026497) & (ws2/.NET CLR Exceptions/# of Exceps Thrown / sec/_Global_ $\geq$ 8.948925) & (ws6/Process/Handle Count/_Total $\leq$ 21258) $\rightarrow$ OTHER (DBclus1/SQLServer:Buffer Manager/Page life expectancy/null $\leq$ 4357) & (DBclus2/Memory/Available MBytes/null $\leq$ 5177) & (ws5/Processor/% Processor Time/_Total $\geq$ 2.104228) $\rightarrow$ OTHER (DBclus1/SQLServer:Buffer Manager/Page life expectancy/null $\leq$ 4088) & (DBclus1/SQLServer:Latches/Average Latch Wait Time (ms)/null $\geq$ 1.02296) & (DBclus2/LogicalDisk/Avg. Disk sec/Write/T: $\geq$ 0.000543) $\rightarrow$ OTHER (DBclus1/LogicalDisk/Disk Reads/sec/W: $\geq$ 20.217216) & (IDws1/Paging File/% Usage/_Total $\geq$ 1.238918) & (ws3/ASP.NET Apps v4.0.30319/Requests Timed Out/__Total__ $\geq$ 1) $\rightarrow$ OTHER (ws6/ASP.NET Apps v4.0.30319/Requests Timed Out/__Total__ $\leq$ 0) & (ws4/Processor/% Processor Time/_Total $\geq$ 13.349845) & (ws1/.NET CLR Exceptions/# of Exceps Thrown / sec/_Global_ $\leq$ 2.83327) & (DBclus1/LogicalDisk/Avg. Disk sec/Write/E: $\geq$ 0.000446) $\rightarrow$ OTHER else $\rightarrow$ LOW
$RULES_{MH}$
(ws3/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ $\leq$ 86408) & (DBclus2/LogicalDisk/Avg. Disk sec/Read/W: $\geq$ 0.000932) & (IDws1/LogicalDisk/Avg. Disk sec/Write/_Total $\geq$ 0.001162) $\rightarrow$ HIGH (ws3/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ $\leq$ 70541) & (DBclus2/LogicalDisk/Avg. Disk sec/Write/W: $\geq$ 0.0005) & (DBclus2/Memory/Page Reads/sec/null $\geq$ 0.046007) $\rightarrow$ HIGH (ws4/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ $\leq$ 81291) & (DBclus1/LogicalDisk/Disk Reads/sec/J: $\geq$ 0.076917) & (DBclus1/SQLServer:Buffer Manager/Page life expectancy/null $\leq$ 131) & (contr1/Server/Bytes Total/sec/null $\leq$ 204.351318) $\rightarrow$ HIGH (ws1/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ $\leq$ 18344) & (ws2/ASP.NET Applications/Request Bytes In Total/__Total__ $\leq$ 7161) & (ws1/Server/Bytes Total/sec/null $\leq$ 2113.22168) $\rightarrow$ HIGH (ws6/ASP.NET Apps v4.0.30319/Request Bytes Out Total/__Total__ $\leq$ 629862) & (IDws2/Memory/Pool Paged Bytes/null $\geq$ 140587008) & (IDws2/Memory/% Committed Bytes In Use/null $\leq$ 19.593651) $\rightarrow$ HIGH else $\rightarrow$ MED

Table 9: Association rules generated during the random sample verification case study

7.3. Analysis Phase

During the analysis phase, 236 PIOs were detected using our approach. For every PIO, the rule coverage matrix was saved into the database so that the covered rules could be manually verified later. In addition, 182 hours were marked as overload hours using the overload detection rule as described earlier in this section. To generate the random sample, we randomly selected 12 starting points for PIOs from the set of PIOs detected using our approach and 5 hours from the set of overload hours from the database.

7.4. Evaluation

The analysis results were evaluated by a performance expert from Exact who has 10 years of experience in performance analysis and deep knowledge of the EOL infrastructure. In this evaluation the expert focused on evaluating a) whether the detected PIO was actually a real PIO and b) whether the rule coverage matrix points to the bottleneck component. To verify whether the rule coverage matrix points to the bottleneck component, the expert used a number of performance reports generated by EOL. These reports contained traditional performance metrics. These reports exhibited the following information for all servers in the system:

- Configuration details for all performance counters (interval, min./max. measurements per hour)
- Details about background services (page views, total duration, query duration, average duration)
- Details about the number of performance counter values monitored versus the number of expected values based on the configuration of the counters
- Details about which servers have the service calls and queries that take the longest to execute
- Details about the running processes (overlap and duration)
- Details per application (duration, query duration)

- Histograms for all performance counters of the average value per hour
- Page views per hour
- Average duration per hour
- Overview of running processes and applications at a certain time

All these reports can be tailored to show only data for a certain period. To decide whether a detected PIO was a real PIO, the expert inspected the reports for variations in these traditional metrics. This process is the usual process for performance analysis at Exact. During the evaluation, the expert:

- Analyzed the performance data monitored around the time of the detected PIO
- Made a manual diagnosis of the system at that time
- Decided whether the detected PIO was actually a PIO
- Compared his diagnosis with the diagnosis made by our approach
- Graded the diagnosis made by our approach with:
  - 0 - (almost) completely wrong
  - 0.5 - partly points in the right direction and/or incomplete
  - 1 - (almost) completely correct

Table **??** and **??** show two examples of this process.

This process was executed for all 17 (12 from our approach and 5 from the average response time rule) detected PIOs in the random sample. Table **??** shows the manual diagnosis, criterium used (rule coverage (RC) or average response time (AVG)) and diagnosis correctness for the complete sample. Note that we did not evaluate the diagnosis quality for 'PIOs' detected with the AVG rule as this rule does not give us a diagnosis. In this case, the 'Detected PIO?' column contains whether our approach detected a PIO during this hour.

Table 10: Example PIO evaluation 1

PIO ID: 1	**Date:** 2012-02-26 01:47:56	**Criterium used:** Rule Coverage

Automated diagnosis:
DBclus2/LogicalDisk/Avg. Disk sec/Write/W:
DBclus2/Memory/Page Reads/sec/null
ws6/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__

Manual diagnosis:
Page reads/sec high on *DBclus2*. Cause: server restarted → cache empty so needs to be filled up.

Verification:
Is real PIO: Yes **Diagnosis correctness:** 0.5
The automated diagnosis is correct but it should point to web servers ws1-ws6 as these are all affected by the restart. Therefore, the diagnosis is incomplete.

Table 11: Example PIO evaluation 2

PIO ID: 2	**Date:** 2012-02-26 02:02:57	**Criterium used:** Rule Coverage

Automated diagnosis:
DBclus2/LogicalDisk/Avg. Disk sec/Read/W:
IDws2/LogicalDisk/Avg. Disk sec/Write/_Total
IDws2/Memory/% Committed Bytes in Use/null
IDws2/Memory/Pool Paged Bytes/null
ws6/ASP.NET Apps v4.0.30319/Request Bytes Out Total/__Total__
ws6/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__

Manual diagnosis:
Several heavy background jobs which were originally scheduled apart from each other are taking much longer now because of database growth, causing them to run at the same time.

Verification:
Is real PIO: Yes **Diagnosis correctness:** 1
The diagnosis is correct as these background jobs are started from web server ws6 and require the identification web server *IDws2* and database cluster *DBclus2* to run.

Table 12: Random sample evaluation

ID	Criterium	Manual diagnosis	Is PIO?	Diagnose quality	Detected PIO?
1	RC	Page reads/sec high on *DBclus2*. Cause: server restarted → cache empty so needs to be filled up.	Yes	0.5	-
2	RC	Several heavy background jobs which were originally scheduled apart from each other are taking much longer now because of database growth, causing them to run at the same time.	Yes	1	-
3	AVG	No PIO.	No	-	No
4	RC	Heavy background job.	Yes	1	-
5	AVG	Yes, a short hiccup due to a load balancer restart causing traffic to be unbalanced.	Yes	-	Yes
6	RC	Heavy background job	Yes	1	-
7	AVG	No PIO.	No	-	No
8	AVG	Combination of 2 and 6.	Yes		Yes
9	RC	Same as 8, but detected by PIO analysis instead of average duration. Diagnosis helps to point correctly to the background jobs but misses a problem on web server *ws5* caused by the background jobs.	Yes	0.5	-
10	RC	Same as 9	Yes	0.5	-
11	AVG	Same as 3	No	-	No
12	RC	Same as 9	Yes	0.5	-
13	RC	Same as 9	Yes	0.5	-
14	RC	Same as 9	Yes	0.5	-
15	RC	No PIO.	No	0	-
16	RC	Problem with a background job which could not connect to an external service, causing it to timeout.	Yes	1	-
17	RC	No PIO.	No	0	-

During the evaluation we noticed that large portions of the detected PIOs were caused by the same events. The most significant event was running the scheduled background jobs during the night. When these background jobs were originally designed, they finished fast due to the smaller database size. Now that the database has grown, these tasks take longer to finish and sometimes their execution overlaps. This causes a slowdown in the system.

Table **??** shows a summary of the results of this case study. The first conclusion we can draw from this table is that our approach has high precision for detecting PIOs (83%). The number of false positives detected by our approach is low, and in fact, it is lower than the number of false positives detected by the average response time rule. In addition, our approach gives a more detailed time frame for the PIO. An example of this is PIO 5 in Table **??** which lasted for approximately 10 minutes. Because the average response time rule notifies per hour, the indication of the time frame is less precise than ours because we notify per minute. However, it is important to realize that simply using the average

response time per minute does not work, because this will lead to a high number of false positives. This is because the long duration of some applications (e.g., report generation) will be emphasized when one minute is used as a time frame, resulting in the detection of a PIO.

In most cases the automated diagnosis using the rule coverage matrix was at least partly correct. In most of these cases, the diagnosis was incomplete. An example of this is PIO 9. In this case, the diagnosis did assist in selecting the background jobs. However, it failed to point out that the CPU usage on web server *ws5* was at 100% for approximately 15 minutes, causing some of the background jobs to slow down. This was noticed by graphing the raw values for the CPU usage on the web servers around the time of the PIO.

After the evaluation, the expert indicated that our PIO analysis approach was effective in assisting during the performance analysis process. Although the expert had access to much information without our approach using the reports, the main problem was that he did not know where to start with the investigation. Our approach helped in providing this starting point.

Table 13: Summary results case study

PIO analysis approach	
PIOs analyzed:	12
Real PIOs (precision):	10 (83%)
Diagnosis quality:	1: 4/12, 0.5: 6/12, 0: 2/12
Average response time rule	
Overload hours analyzed:	5
Real overload (precision):	2 (40%)
Correct classification by PIO analysis approach:	5 (100%)

8. Discussion

8.1. The Requirements Revisited

8.1.1. Requirement 1: Detect the timeframes during which the system performed relatively slow.

In our evaluation in Section **??** we have shown that our approach is capable of detecting PIOs with a high precision. Initially, we aimed at detecting the start and end time of a PIO. In practice however, together with the expert we found that the end time of a PIO is difficult to determine. The reason for this

becomes clear from Figure **??**, in which there are 4 peaks around $t = 400$. The question is whether these 4 peaks represent 1 or 4 PIOs. In addition, during the case study we noticed that the expert intuitively combined PIOs that lie closely together in his investigation, rendering this question unimportant. Therefore, we decided to use only the PIO starting time.

8.1.2. Requirement 2: Detect the component(s) that is/are the bottleneck component(s).

In our evaluation in Section **??** we have shown that our approach is successful in diagnosing a bottleneck in many cases. It was especially successful in detecting problems with recurring tasks, due to the fact that it is easy to find patterns in PIO times in this case. Especially in combination with information from the application log (e.g., running applications and/or tasks during the PIO), the expert was capable of completing his investigation for performance optimization.

However, it appears difficult to diagnose several bottlenecks at the same time. The main cause for this is the quality of the association rules. These rules should exhibit as much information about performance counters as possible. Because the rule generation is automated, it is possible that rulesets for some training periods are not as detailed as desired. Therefore, a possibility for improving the quality of the rulesets is to use several training periods and combine the resulting rulesets. This possibility will be addressed in detail in future work.

8.2. Automatability & Scalability

Automatability. All steps in our approach are automated. An interesting problem is when to update the association rules. In the EOL proof-of-concept we have shown that 3 months after training, our rulesets were still able to estimate the `SARatio`, which leads to the expectation that the rules do not need regeneration often. An example of a situation in which the rules need to be regenerated is after removing or adding a new server to the system. Our current solution is to retrain all the rules with the new set of performance counters.

In our current case studies the length of the period during which training data was monitored was based on the availability of the data. In future work we will address the challenge of finding the ideal training period.

Scalability. The `PerformanceLog` and `ApplicationLog` analyzed during the case study in Section **??** contained respectively 28 million and 135 million records. Preparing the data and training the association rules took approximately 10 minutes. 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.

Limitations. 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.

8.3. Different Applications

An application which lies closely to our purpose of finding the moments when the system performs relatively slow is anomaly detection. The difference between a performance anomaly and a PIO is that the occurence of an anomaly is incidental, while the occurence of a PIO is structural. While our approach is capable of detecting performance anomalies, it is important to realize that it is based on supervised learning. Supervised learning has inherent limitations for anomaly detection, since a classifier trained with supervision can only detect anomalies which have been seen before or are similar to earlier events. The problem with anomalies is that they often have not occurred before, making it difficult to detect using supervised training. Therefore, our approach is suitable for detecting some performance anomalies but we expect a high number of false negatives.

Another interesting application of our approach is that it can be used, after some extension, in regression testing to validate a baseline performance after

updates. Because our approach is trained with the assumption that approximately 5% of the system is running relatively slow, we can use this assumption to roughly validate the performance of the system after an update. If our approach detects PIOs for more than 5% of the time, we know that the performance of the system has gotten worse and we need to analyze exactly what part of the update causes this.

8.4. Comparison With Other Techniques

We have shown in our evaluation that our approach is more precise than using an average response time threshold. In addition, it gives a more detailed indication of the starting time of a PIO. Likewise, we expect our approach outperforms the use of thresholds for other traditional metrics, because these do not take user and application characteristics into account as described in Section **??**.

Another important advantage of our approach over other techniques is that it contains temporal information. The advantage of having access to temporal information is that the in the diagnosis we can emphasize performance counters which occurred throughout the PIO. These counters are more likely to give an accurate bottleneck diagnosis. The rule coverage matrix allows experts to give priority to certain performance counters in their analysis depending on their value in the matrix. For example, in Figure **??**, *S2PC1*, *S2PC2* and *S5PC2* would more likely be interesting for investigation than *S1PC1* and *S5PC1*.

8.5. Lessons Learned

Initially, we expected that the expert would be most interesting in longer lasting PIOs, as these are more likely to yield greater improvements when exploited. However, during the evaluation we found out that he was especially interested in the shorter lasting PIOs. The main reason was that these shorter PIOs must usually be exploited by subtle performance improvements, making them more difficult to spot with the naked eye. In addition, it is usually easier to diagnose longer lasting PIOs, because their is more information available. The lack of information makes shorter lasting PIOs more challenging to analyze.

In addition, we found during the evaluation that the intensity transformation does not work well in practice. The main reasons for this are:

- Because the transformation uses a sliding window, the PIO possibly has already been running for some time. The expert wanted immediate notification when a PIO started.
- The downward part of the intensity graph is confusing as the PIO is actually already over at that time. This was the reason to use only the starting time of a PIO as mentioned in Section **??**.

These limitations must be taken into account when using the intensity transformation.

9. Threats to Validity

9.1. External Validity

We acknowledge that 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. While the RUBiS case is not representative for modern applications anymore [**?**], it is a widely-used benchmark in performance studies and a useful second validation of our approach.

Only one expert was used for the evaluation during the case study. Because our approach yields a result which is open to different interpretations, this evaluation is subjective. Therefore, the evaluation of our approach by the expert is subjective. However, in our opinion the evaluation is valid as this expert has many years of experience with performance maintenance and the case study system.

In our case study we have evaluated only a sample of the automated analysis results. Because this sample was selected randomly we expect it is representative of the complete result set.

9.2. Internal Validity

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 proof-of-concept of our `SARatio` estimation approach.

A possible threat to validity is the fact that the overhead introduced by monitoring the performance counters influences our training set and therefore our classification scheme. However, as accessing performance counters is relatively cheap [**?**], we assume that reading the value of n performance counters will have *O(n)* overhead for every time period we make a measurement. Because this results in constant overhead for all measurements, we assume that the overhead introduced in the training set will also exist for the measurements made during the classification phase and will therefore be negligible.

10. Related Work

In previous work, we have done a preliminary evaluation of our approach by conducting a contextual interview with performance experts from industry [**?**]. Feedback elicited during this interview led to the approach as it is presented in the current paper. The rest of this section discusses methods for assisting performance experts in finding performance improvement opportunities.

Performance Anomaly Analysis. Important tools for performance experts are anomaly detection mechanisms. Often, these mechanisms detect anomalies that can be prevented in the future by improving the performance of the system.

Breitgand et al. [**?**] propose an approach for automated performance maintenance by automatically changing thresholds for performance metrics for components, such as response time. 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 predict and adapt the thresholds for components

such that the true positive and negative rate converge to their threshold, hence improving the performance of the system. In contrast to our work, they use single threshold values for performance metrics, while we use association rules which lead to combinations of thresholds.

Cherkasova et al. [**?**] present an approach for deciding whether a change in performance was caused by a performance anomaly or a workload change. They create a regression-based model to predict CPU utilization based on monitored client transactions. Zhang et al. [**?**] do anomaly detection by forecasting a value for CPU utilization and comparing it to the actual utilization. In case the difference is significant, an anomaly is detected. Zhang disregards administrative tasks but our approach takes these into account. While Cherkasova et al. and Zhang et al. focus on CPU utilization, our approach takes more metrics into account.

Correa and Cerqueira [**?**] 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, their work focuses on distributed systems, making network traffic an important part of the equation.

In Oceano, Appleby et al. [**?**] correlate metrics such as response time and output bandwidth with SLO violations. Oceano extracts rules from SLOs in order to create simple thresholds for metrics. In contrast, our approach uses more detailed performance metrics and more complex thresholds.

Munawar et al. [**?**] search for invariants for the relationship between metrics to specify normal behaviour of a multi-tier application. Deviations from this relationship help system administrators to pinpoint the faulty component. In their work they use linear regression to detect relationships between metrics, which limits their research to linear relationships. Our approach does not explicitly look for direct relationships between metrics, but focuses on combinations of values instead.

Cohen et al. [**? ?**] present an approach to correlate low-level measurements

with SLO violations. They use tree-augmented naive Bayesian networks as a basis for performance diagnosis. Their work is different from ours in the way we detect the possible performance improvement. As we combine several rules, our approach is capable of giving a more detailed analysis of the location of the improvement.

Syer et al. [**?**] use covariance matrices to detect deviations in thread pools that indicate possible performance problems. The focus of their approach is on thread pools while ours is not limited to a particular architectural pattern.

Malik et al. [**?**] have presented an approach for narrowing down the set of performance counters that have to be monitored to automatically compare load tests by using statistics. Their technique also ranks the performance counters based on their importance for load tests. Their work focuses on selecting metrics (i.e., the dimension reduction problem), while our work focuses on analyzing those metrics instead.

Jiang et al. [**?**] analyze log files to see if the results of a new load test deviate from previous ones. This allows developers to analyze the impact of their changes. Nguyen et al. [**?**] address a similar problem, namely the problem of finding performance regressions. The focus of these approaches is on analyzing whether a change had the desired effect on performance, while our approach focuses on finding what to change.

Profiling. Profilers are tools which collect run-time information about software [**?**], such as the amount of memory used or the number of instructions executed. More advanced profilers analyze the 'run-time bloat', e.g., unnecessary new object creations [**?**]. Profilers assist system administrators in the way that they help identify the block or method which uses the most resources and hence may form a bottleneck.

Agrawal et al. [**?**] use dynamic analysis to count the number of times basic blocks are executed. They define the blocks that are executed most as possible bottlenecks and hence try to optimize those blocks.

Bergel et al. [**?**] extend profiling with the possibility to detect opportunities for code optimization. Using visualizations, they advise developers on how to

refactor code so that it will run faster. Their advice is based on principles such as making often called functions faster.

In general, while there are methods for decreasing the amount of data and instrumentation [**?** **?**], execution profiling introduces considerable overhead due to the large amount of data that needs to be monitored. In addition, because profilers usually analyze *hot* code (e.g., the code that uses the most CPU cycles), they are not always directly suitable for detecting all possible performance improvements [**?**]. Finally, it is possible that many sites must be monitored in a distributed environment. Therefore, while execution profiling plays an important role in performance maintenance, its use should be minimally. Our approach can assist in reducing the execution profiling overhead by pinpointing the hardware on which profiling should be done.

LagHunter [**?**] tries to decrease the overhead by only profiling *landmark* functions, methods of which the human-perceptible latency can become too high. LagHunter implements a method for automatically selecting which functions are landmark functions. These landmark functions are considered possible performance issues as they heavily influence the human-perceptible latency and therefore can become an annoyance for users.

Using Heat Maps for Performance Maintenance. Heat maps have been used for performance analysis before [**?** **?**], but we have evaluated our approach in an industrial setting and on multi-server data. In addition, in previous work heat maps were used to plot the raw values of performance counters, without the addition of extra information to assist the performance expert. Our approach for heat maps does include this extra information. Heat maps have also been used in other areas, e.g., repository mining [**?**].

11. Conclusion

In this paper we have proposed a technique for detecting and analyzing performance improvement opportunities (PIOs) using association rules and performance counter measurements. We have proposed the `SARatio` metric, which allows us to specify the starting point of a PIO more precisely than traditional

metrics. We have shown that this metric can be estimated using performance counter values in a proof-of-concept case study on a synthetic benchmark and an industrial application.

In addition, the results of our PIO analysis approach were manually verified in an industrial case study by a performance expert. The results of this case study show that our approach has high precision when detecting PIOs and can assist performance experts in their investigation of possible performance optimizations. In short, our paper makes the following contributions:

- An approach for detecting and analyzing PIOs using association rules, performance counters and the `SARatio` metric.
- A proof-of-concept case study in which we show that the `SARatio` can be estimated using association rules and performance counters.
- An evaluation of our approach for PIO analysis done by a performance expert.

Revisiting our research questions:

- **RQ 1** *How can performance counter values provide assistance during the performance optimization process?* We have presented our approach for PIO detection, which is based on performance counter values and provides assistance during the performance optimization process. We have shown in two case studies that this approach is accurate and improves the speed with which performance experts can do their investigation for performance optimization.
- **RQ 2** *How can we interpret these values so that they can lead to the identification of the bottleneck component(s) of a system?* We have presented an approach for PIO analysis using the rule coverage matrix. We have shown in an industrial case study that the results of this approach are accurate and assist the performance expert in detecting the bottleneck component(s).

11.1. Future Work

In future work we will focus on selecting the most suitable training period or a combination of training periods in order to increase the quality of the association rules. In addition, we will investigate the possibilities of using multiple models.